INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

INTEGRATION OF MOBILE AGENT ENVIRONEMT WITH LEGACY SNMP

By

Yian Wang, B. Eng.

A Thesis Submitted to
the Faculty of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

August 14, 1998

(c) Copyright 1998, Yian Wang

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-36899-8

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of this thesis

**“INTEGRATION OF MOBILE AGENT ENVIRONEMT
WITH LEGACY SNMP”**

Submitted by Yian Wang, B. Eng.
in partial fulfillment of the requirements
for the degree of Master of Engineering

Department Chair

Thesis Supervisor, Dr. Bernard Pagurek

Carleton University

August, 1998

ABSTRACT

Managing communication networks based on mobile code is an innovative approach. Integration of the mobile code environment (MCE) with legacy SNMP systems is of great importance. This thesis has analyzed the integration requirements, and designed and implemented an eXtensible Mobile Supported (XMS) SNMP agent to support various communication requests from the MCE of Perpetuum Mobile Procura Agent Project.

The XMS-SNMP agent has been built by modifying the public UCD-SNMP agent to support two lightweight communication protocols, including DPI and RDPI, and a management MIB. The DPI protocol can be used by a mobile agent to dynamically extend MIB groups to the XMS-SNMP agent, while the RDPI protocol can be used by a mobile agent to access MIB variables in the XMS-SNMP agent. The management MIB can be used by a network management system to query and maintain those dynamically registered mobile agents and MIB groups.

On the application side, a set of DPI APIs and a set of RDPI APIs have been designed and implemented in Java. Based on the DPI APIs, a DPI application framework has been provided to further simplify the implementation of DPI protocol applications. Based on the RDPI APIs, an RDPI tool set has been provided for users to access MIBs in the XMS-SNMP agent from a command line. Furthermore, a thorough test plan has been developed to test the XMS-SNMP agent in a systematic way. Various tests have been conducted to ensure the correctness and robustness of the XMS-SNMP agent.

In the integration protocol area, the RDPI protocol has been defined by modifying a subset of data messages of the DPI protocol. The simplicity of the RDPI protocol has been demonstrated by analytically comparing it with the SNMP protocol. In addition, the protocol mismatch problem between the CMU/UCD-SNMP and the DPI in some SNMP *set* situations has been uncovered, fully analyzed, and solved by the addition of an *action* packet type to the DPI protocol.

ACKNOWLEDGEMENT

First of all I would like to express my grateful thanks to my thesis supervisor Dr. Bernard Pagurek for his guidance and patience throughout his supervision of this work. The financial assistance from TRIO and NSERC is also greatly appreciated.

I would like to express my grateful thanks to Dr. Chinneck for his support, encouragement, and guidance during the first half of my studies.

I would also like to extend my appreciation to Dr. Andrezej Bieszczad, Mr. Tony White, and the members of the Network Management Laboratory for the weekly constructive and informative discussions on mobile agents.

I would like to thank Mr. Narendr Mehta, and Mr. John Knox, for maintaining ARTIE, the host the SNMP agent is installed on, in service throughout the work.

Finally, a special thank you to my mother, my daughters, and my husband whose love, patience, and support during the years of my study have enabled this work to reach fruition.

TABLE OF CONTENTS

Abstract	iii
Acknowledgment	iv
List of Figures	viii
List of Tables	xii
List of Acronyms	xiii
Chapter 1: Introduction	1
1.1 Problem Overview	1
1.2 Thesis Contributions	5
1.3 Thesis Outline	6
Chapter 2: Functional Requirement and Architecture	9
2.1 Mobile Agent Environment Overview	9
2.2 MCE and SNMP Agent Communication Requirements	11
2.3 XMS-SNMP Agent Functional architecture	18
Chapter 3: Integration Protocols	20
3.1 RDPI Protocol	20
3.2 DPI Protocol Enhancement	25
Chapter 4: XMS-Agent Design and Analysis	28
4.1 CMU-SNMP and UCD-SNMP Implementations	28
4.2 XMS-SNMP Agent Data Model	34
4.3 XMS-SNMP Agent Message Scheduling	40
4.4 SNMP Request Processing	42
4.5 DPI Request Processing	44
4.6 RDPI Request Processing	47

Chapter 5: Mobile Agent Management	50
5.1 Mobile Agent Management Analysis	50
5.2 Management MIB	53
5.3 Management MIB Access Functions	58
5.4 Integration of the Management MIB Access Functions with Agent	61
Chapter 6: DPI API, AFW, and Applications	64
6.1 DPI Application Program Interface	64
6.2 DPI Application Framework	76
6.3 MIBGroup Subclass Sample	81
6.4 DPI Subagent Sample	88
Chapter 7: RDPI API and Applications	91
7.1 RDPI API	92
7.2 RDPIInterface Class	98
7.3 RDPI Applications	99
Chapter 8: Test Design	104
8.1 Test Areas	104
8.2 Test Strategy	106
8.3 Test Cases	111
Chapter 9: Conclusions	119
11.1 Summary	119
11.2 Future Work	120
References	121
 Appendix A: RDPI Protocol And SNMP Protocol Comparison	 123
A.1 SNMP Message Definition	123
A.2 SNMP Message Encoding	128

A.3	SNMP and RDPI Overhead Comparison	131
A.4	SNMP and RDPI Performance Comparison	134
Appendix B: THE SNMP and DPI Mismatch Problem Further Discussion		141
B.1	Test Configuration and Result	141
B.2	UCD-SNMP <i>SET</i> Request Processing Analysis	142
B.3	The SNMP <i>SET</i> Analysis Before the Addition of <i>ACTION</i>	144
B.4	The SNMP <i>SET</i> Analysis After the Addition of <i>ACTION</i>	151
Appendix C: Mobile Agent Management MIB Definition		153
Appendix D: XMS-SNMP Agent Configuration		164
Appendix E: DPI and SNMP Object Type Conversions		167

List of Figures

Figure		Page
2.1	Mobile Code Environment Infrastructure	10
2.2	Mobile Agent Talks to SNMP Agent through VMC	11
2.3	Mobile Agent Extends SNMP MIB	12
2.4	Mobile Diagnostic Agent Accessing SNMP MIB	13
2.5	Mobile Diagnostic Agent Accessing Extended SNMP MIB	14
2.6	Mobile Agents Extend and Access SNMP MIB	15
2.7	MMVA Access Extended SNMP MIB	17
2.8	XMS-SNMP Agent Functional Architecture	18
4.1	CMU-SNMP Agent Data Model	29
4.2	CMU-SNMP “subtree” Structure	30
4.3	CMU-SNMP “variable” Structure	30
4.4	CMU-SNMP MIB Access Function Interface Definition	31
4.5	XMS-SNMP Agent Data Model	35
4.6	XMS-SNMP Agent “subtree” Structure	36
4.7	XMS-SNMP Agent “registerTree” Structure	37
4.8	XMS-SNMP Agent “subagent” structure	39
4.9	socket list and socket number definitions in snmpd.c file	40
4.10	XMS-SNMP Agent Message scheduling	41
5.1	Directly Managing Mobile Agents through SNMP	51
5.2	Managing Mobile Agents through VMC	52
5.3	Managing MCE through SNMP	53
5.4	The Position of the XMS-SNMP Agent Management MIB	54

5.5	"subagentMIB" Structure	55
5.6	"registerTreeMIB" Structure	57
5.7	Subagent and Register Tree Variables	61
5.8	Built-in "subtree" Table	62
6.1	DPI Packet Classes Hierarchy	65
6.2	DPI Application Framework	76
6.3	MibGroupSample Constructor	82
6.4	MibGroupSample "get" Method	83
6.5	MibGroupSample "getnext" Method	84
6.6	MibGroupSample "set" Method	85
6.7	MibGroupSample "commit" Method	86
6.8	MibGroupSample "action" Method	87
6.9	MibGroupSample "undo" Method	88
6.10	DPITest.java Code List	89
7.1	RDPI Packets, Interface, and Application	91
7.2	RDPI Packet Class Hierarchy	92
7.3	RDPIGet Tool Code List	100
7.4	RDPIGetnext Tool Code List	101
7.5	RDPISet Tool Code List	102
8.1	Test Configuration	106
8.2	Subagent Sample 1 Attributes	107
8.3	Subagent Sample 2 Attributes	107
8.4	Access Extended MIB and Management MIB fro RDPI tools	109

8.5	“unregister” and “close” through management MIB	109
8.6	Trigger subagents to send request scenario	110
A.1	SNMP Protocol	124
A.2	SNMP Message in ASN.1	125
A.3	SNMP PDU in ASN.1	125
A.4	SNMP VarBindList and VarBind in ASN.1	126
A.5	SNMP ObjectName and ObjectSyntax in ASN.1	126
A.6	SNMP SimpleSyntax and ApplicationSyntax in ASN.1	127
A.7	SNMP Basic Object Types in ASN.1	127
A.8	BER encoding structure	128
A.9	BER Tag Field	129
A.10	BER Length Field	130
A.11	SNMP Set packet encoding	132
A.12	Processing "SEQUENCE OF"	135
A.13	asn_parse_header() Function Call	135
A.14	ASN_EXTENSION_DI Macro Definition	136
A.15	asn_parse_length() Function Call	136
A.16	Processing "SEQUENCE"	137
A.17	Processing "OBJECT IDENTIFIER"	137
A.18	asn_parse_objid() Function Call	138
A.19	copy() Function Call	139
B.1	DPI Protocol Test Configuration	142
B.2	A read-write variable definition in a subagent	147
B.3	read-write variable change after receiving DPI a <i>SET</i>	147

B.4	read-write variable change after a DPI <i>COMMIT</i>	149
D.1	Agent Configuration File Structure	164
D.2	cshrc file	165
D.3	sysgroup.ini	165
D.4	vltrap.dst	165

List of Table

Table		Page
3.1	RDPI Packet Header	20
3.2	RDPI <i>GET/GETNEXT</i> Packet	21
3.3	RDPI <i>SET</i> Packet	22
3.4	RDPI <i>RESPONSE</i> Packet	23
3.5	Mapping SNMP <i>SET</i> operations to DPI <i>SET</i> Operations	26
8.1	DPI Messages for Testing	104
8.2	RDPI Message for Testing	105
8.3	XMS-SNMP Agent Management MIB Variables to Test	105
A.1	RDPI <i>SET</i> Packet Overhead Calculation	133
E.1	DPI and SNMP Object Type Conversion	167

List of Acronyms

API	application program interface
AFW	application framework
ASN.1	Abstract Syntax Notation One
BER	basic encoding rules
CMIP	common management information protocol
CMU	Carnegie-Mellon University
DPI	distributed protocol interface
eSNMP	extensible SNMP agent
ID	identifier
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
JVM	Java Virtual Machine
MAE	mobile agent environment
MC	mobile code
MCD	mobile code daemon
MCE	mobile code environment
MCM	mobile code manager
MDA	mobile diagnostic agent
MIB	management information base
MF	migration facility
MMVA	mobile meta-variable-agent
NC	network component
NE	network element
NMS	network management system

OID	object identifier
OSI	open systems interconnection
PDU	protocol data unit
RDPI	Reverse DPI
RFC	request for comments
SMI	Structure of Management Information
SMUX	SNMP Multiplexing
SNMP	Simple Network-Management Protocol
SNMPv2	Simple Network-Management Protocol version 2
TCP	transmission-control protocol
UDP	user datagram protocol
UCD	University of California at Davis
VMC	virtual managed component
XMS-SNMP	extensible mobile supported SNMP

CHAPTER 1

INTRODUCTION

1.1 Problem Overview

The telecommunication networks in service today are a composite of heterogeneous network elements which are from different vendors. Service interworking is through standard protocols with which all vendor's equipment must comply. Network management systems are based on manager-agent, or client-server model. In this model, each network element has a dedicated agent which performs manager's requests and responds to manager's query on managed resources through standard protocols. Managed resources of a network element are represented by device-independent MIBs based on either IETF standard or OSI standard. Accordingly, the manager-agent communication protocols are either SNMP of IETF standard[7] or CMIP of OSI standard[15]. Through the standardized MIBs and the manager-agent protocols, a third party network management system can access any managed resources of different types of network elements from different vendors.

However, there are still problems existing in the current network management model. The first problem is that a network operator usually has to deal with a proliferation of human-machine interfaces and interoperability problems[1]. This is because some vendors implement their own management systems for their own network equipment. It is very difficult to build a single management system or a uniform interface of multiple management systems to manage all different network elements from different vendors. Hence, a network service provider usually ends up with many different management

systems.

The second problem is that an agent of a network element can only supply a huge amount of “raw data”, and has no intelligence. To obtain useful information, for example to localize a fault, the network manager has to pull substantial data from the agent remotely and to process them in the management station. This adds a considerable strain on the throughput of the network[1].

The third problem is related to the flexibility and extensibility of the network management. As long as the systems, including both the manager side and the agent side, are built, it hardly adds any new features without substantial rework on both sides.

There are many ways to address the above problems. The first problem can be solved by using Web/Java technology. On the agent side, every network element needs to implement its agent software using Java. The functions of the Web agent are application-dependent. Vendors are free to use any creative ways to present their managed resources. On the management side, it is not necessary to implement vendor specific software. All that is required is to design a vendor-independent and device-independent framework which has links to the web agents. By invoking the links, network operators can download the web agent codes to the local management station and execute them in the Java browser locally. It is obvious that different vendor devices can be integrated and managed from the web browser much more easily than the current implementations. There is no specialized management software necessary to configure or monitor a device in the management station. The network management stations are not only platform-independent but also location-independent[11].

The second problem can be solved by using the concept of “Delegation of authority”[19]. Instead of pulling huge amount of data from a network element and processing the data in the management station, a delegated agent can be sent to the network element to collect and process data locally. Therefore, the requirement for data

transmission between the management station and the agent is reduced, and overall network throughput is increased. Using distributed network management can also solve this problem. In this approach, middle-level management systems are built to pull and process data from a number of local network elements. This avoids the heavy data transfer from agents to the central management station.

The third problem can be partially solved by using the concept of “agent extensibility”. The idea is to partition the original monolithic SNMP agent into one master agent and zero or more subagents. The master SNMP agent is responsible for receiving management requests and sending management responses. The subagents are responsible for implementing various parts of the MIBs. Extended MIB can be dynamically registered with the master agent at run-time. The management station can access any part of the MIB transparently. “partially solved” means that this only solves the MIB extensibility problem on the agent side. On the management station side it is still necessary to do substantial amount of work to get the extended MIB integrated with the rest of the applications.

Several protocols have been developed for agent extensibility. Among these protocols, SMUX[14], or SNMP multiplexing, was the pioneer protocol developed in 1990. But one of the problems with SMUX is its performance. Since the core protocol between the agent (master) and its peers (subagents) was SNMP, the subagent needs to fully understand the heavy SNMP protocol. The DPIv2 [18] protocol has attracted most of the attention. Its advantage is the lightweight messages between the master SNMP agent and the subagents. The eSNMP[9], an extensible SNMP agent protocol, is a variation of the DPI developed at Digital in 1995. It followed the basic DPI model, but with some changes specifically for Digital UNIX (formerly known as DEC ALPHA OSF/1). For example, they use UNIX Domain Sockets instead of network domain sockets. The AgentX[10] is an evolved agent extensibility protocol from DPI. The most prominent difference between the AgentX and DPI is that the AgentX supports a “region of MIB” registration while the DPI

only supports a sub-tree registration. A “region of MIB” is defined as a fully-qualified instance name, or a partial instance name or a MIB table, or ranges of any of these.

The most advanced idea for network management is to use mobile agent[1][2][6]. Basically, a mobile agent is an independent piece of software that travels in the network and accomplishes tasks remotely. For example, a mobile agent can be sent to a network for device discovery. Although the mobile agent approach addresses the above problems as described in [1], the significance of this approach is its ambition of building a self-maintained, error-free, plug-and-play network where mobile agents can take care of many aspects of network management tasks such as configuration and maintenance.

Despite the fact that there are problems existing in today’s network management model and that there are better ideas to solve the problems as described above, SNMP continues to play a major role in the network management marketplace and it will not die [5]. As stated in the article of “*Some experiences with emerging management technologies*[5]” from Cisco Systems: “The SNMP SMI is what saves SNMP”. It is further explained that “what we have learned over the years is that MIB definitions serve us well as ‘simple to understand contracts’ between device implementors and application implementors’ Better ideas than SNMP come and go. Most fail because they don’t realize SNMP represents a pretty nice balance between the possible and the practical”. In addition, SNMP has gained wide acceptance and deployment which make it impossible to throw away the current SNMP based network management systems in the near future.

On the other hand, accessing management resources is a basic requirement for any kind of management model. The simple way of representing management resources by a Management Information Base (MIB) might be a good candidate for any kind of next generation network management system. It is very reasonable to believe that the existing network management based on SNMP will co-exist with the next generation management system in the future. Therefore, the interworking between the two systems is on avoidable.

This thesis is devoted to analyzing and solving the problems of integrating the advanced network management model using mobile agents and the current SNMP-based network management model. It is part of the Perpetuum Mobile Procura Agent Project[12] with the objective of integrating the mobile code environment (MCE) of [12] with the legacy SNMP. Specifically, it is focused on how to modify/extend legacy SNMP agents to support the most advanced network management paradigm using mobile agents.

1.2 Thesis Contributions

This thesis has made several contributions. The first contribution is the design and implementation of the eXtensible Mobile Supported (XMS) SNMP agent. The XMS-SNMP agent is developed by modifying the public UCD-SNMP[22] agent to support two lightweight protocols (DPI[18] and RDPI) and a management MIB. The DPI protocol is used for a mobile agent to dynamically add MIB groups to the XMS-SNMP agent, while the RDPI protocol is used for a mobile agent to access the MIB variables in the XMS-SNMP agent. The management MIB is used to maintain the dynamically registered mobile agents and MIB groups. It can be used to clean orphaned objects in the XMS-SNMP agent due to abnormal connection interruptions between the XMS-SNMP agent and its mobile subagents. The way a network management system (NMS) communicates with the XMS-SNMP agent using SNMP protocol remains the same.

The second contribution is made on the application side of the XMS-SNMP agent. It has provided a set of DPI application program interfaces (APIs) and a set of RDPI APIs in Java. Furthermore, it has implemented a DPI application framework (AFW). By using the DPI APIs and AFW, an application developer only needs to implement application-specific MIBs. It has also implemented a set of command-line RDPI tools which can be used to access MIB variables in the XMS-SNMP agent. Besides, a thorough test plan has been developed to test the XMS-SNMP agent and the APIs to ensure the correctness and

robustness of the design and implementation.

The third and the final contribution is made in the integration protocol area. It has defined the RDPI protocol based on the DPI protocol. The advantages of RDPI over the SNMP protocol are analytically compared. It has also enhanced the DPI protocol to solve the protocol mismatch problem between the DPI and the CMU/UCD SNMP for the SNMP *set* request. The solution is the addition of an *action* packet type to the DPI protocol. The test scenarios that uncovered the problem has been fully described and analyzed.

1.3 Thesis Outline

In the remainder of this thesis, Chapter 2 thoroughly analyzes the requirements for extending the legacy SNMP agent to support the Mobile Code Environment (MCE). Based on the requirement analysis, the functional architecture of the eXtensible Mobile Supported SNMP agent (XMS-SNMP agent) has been proposed. The XMS-SNMP agent is a SNMP agent with two additional interfaces, namely DPI and RDPI, and a management MIB.

Chapter 3 defines the RDPI protocol and describes the DPI protocol enhancement which is used to solve the protocol mismatch problem between the DPI and SNMP. More detailed protocol issues are discussed in Appendix A and B. Appendix A analytically compares the RDPI protocol and SNMP protocol to demonstrates the simplicity of the RDPI protocol. Appendix B describes the test scenario that uncovered the protocol mismatch problem between the UCD-SNMP agent and the DPI protocol for SNMP *set* request. It further analyzes of the SNMP *set* processing in collaboration with the DPI protocol which provides the rationale for the addition of the *action* packet type to the DPI protocol.

Chapter 4 presents the design and implementation of the XMS-SNMP agent. First, it analyzes the pertinent data models of the CMU-SNMP and UCD-SNMP agent implementations. Secondly, it describes the data model of the XMS-SNMP agent based on

the UCD-SNMP agent. Finally, the dynamic behavior of the data model is fully analyzed by applying different messages from different interfaces, including DPI, RDPI, and SNMP.

Chapter 5 describes the design and implementation of the management-MIB for mobile agent. The purpose of the management-MIB is to maintain the dynamically registered mobile subagents and extended MIB groups. In abnormal situation where the connections between the agent and its mobile subagents are lost, the orphaned objects can be cleaned through this management-MIB. Appendix C provides the XMS-SNMP agent's management-MIB definition which is useful during maintaining and testing the XMS-SNMP agent.

Chapter 6 provides a set of object oriented DPI APIs, application framework, and some application samples. The DPI APIs hide the DPI packet level details. The application framework, consisting of a *DPIInterface*, a *DPISubagentEngine*, and one or more *MIBGroups*, further hides more DPI-related details and builds an infrastructure for applications. By using these APIs and application framework, users are required to only extend the MIB Group class to implement application-specific MIBs. The samples demonstrate how to build a subclass of the *MIBGroup* class and how easy it is to build a DPI subagent by using only a few lines of code.

Chapter 7 provides a set of object-oriented RDPI APIs for mobile agent applications to access MIBs in the XMS-SNMP agent or subagents. By using these RDPI APIs, mobile agent developers are not required to know the implementation details of the RDPI protocol. A set of RDPI command-line tools has been implemented using the RDPI APIs. Users can use this tool set to access MIB variables in the XMS-SNMP agent or subagents.

The codes in Chapter 6 and Chapter 7 are written in Java. The reason for using Java is that the *Perpetuum Mobile Procura Project* has chosen Java to implement its platform-independent mobile code environment, and the work of this thesis is a part of the

infrastructure.

Chapter 8 describes the test plan which covers test strategy and various test scenarios. Considering that this work is part of the MCE infrastructure and future applications will use the XMS-SNMP agent and the APIs, a great effort has been made to make the XMS-SNMP agent and the APIs correct and robust. The test scenarios can also serve as a user guide to the system.

Chapter 9 summarizes the thesis and describes two major research directions which arise during the course of this work.

Appendix D provides a user guide with information about the XMS-SNMP agent configuration, start, and preliminary verification. Appendix E provides a conversion table which lists the SNMP object types and their corresponding DPI object types for the convenience of references.

CHAPTER 2

INTEGRATION REQUIREMENTS AND FUNCTIONAL ARCHITECTURE

The objectives of this chapter are to identify the requirements and to define the functional architecture for the integration of the mobile code environment (MCE)[12] with the legacy SNMP agent. Section 2.1 briefly introduces the MCE of Perpetuum Mobile Procura Agent Project [12] with emphasis on the portion related to the integration problem. Section 2.2 analyzes the integration requirements by applying various types of communications between the MCE and the legacy SNMP. Based on the derived requirements, section 2.3 defines the eXtensible Mobile Supported (XMS) SNMP agent, which can best support all types of communications required by the MCE.

2.1 Mobile Code Environment Overview

The MCE infrastructure of Perpetuum Mobile Procura Agent Project is shown in Figure 2.1. The assumption is that each Network Element (NE) is Java-enabled and the infrastructure is implemented in Java. The Java code is compiled into byte-code and is run in the Java Virtual Machine (JVM). The main reasons for choosing Java are Java's portability and built-in facilities to shift code to remote locations for remote execution [1].

The principal component running inside the JVM is the Mobile Code Daemon (MCD). The MCD is Java code transport facility which listens on two ports (UCD and TCP) for incoming messages. Other MCDs on other NEs can send messages to the ports. Java code mobility is achieved by containing the Java byte-code in the messages. Upon the reception of the byte-code of an mobile agent, the MCD validates the object classes,

instantiates the objects, and then starts the agent by calling the agent's start up code if applicable. After completion of its task, the mobile agent can ask the MCD to migrate it to another NE. The Migration Facility (MF) is used to determine destinations for the agent. The Mobile Code Manager (MCM) is used to manage mobile agents.

Figure 2.1 Mobile Code Environment Infrastructure

The mobile code does not directly access managed resources. It accesses managed resource through a software entity called Virtual Managed Component (VMC).

It is also suggested that interworking between the MCE with the legacy SNMP systems go through the VMC. One example given in [1] is that both the VMC and the SNMP agent have DPI protocol implemented, and the VMC is registered as a subagent to SNMP agent, providing data in response to SNMP requests. This scenario is shown in Figure 2.2.

Figure 2.2 Mobile Agent Talks to SNMP Agent through VMC

2.2 MCE and SNMP Agent Communication Requirements

The way a mobile agent communicates with the SNMP agent is application dependent. From a SNMP agent's point of view, there are three types of communications between the MCE and the SNMP agent. The following subsections will describe each of them to derive the requirements that the legacy SNMP agent should support.

2.2.1 Mobile Agents Extend the Legacy SNMP Agent MIB

This type of communication is that a mobile agent is used to dynamically extend the legacy SNMP agent MIB. This type of mobile agent plays the role of a subagent of the SNMP agent, and can be called a *mobile subagent* (MSA). A MSA communicates with the SNMP agent in the same way that a subagent communicates with the SNMP agent as in the agent extensibility area[18]. Therefore, some agent-extensible protocols can be used to support this type of communication.

Figure 2.3 shows the communication messages between a mobile subagent and a

legacy SNMP agent using the DPI protocol. These messages can be categorized into three groups: control, data, and trap. The “control messages” are used for two purposes. One purpose is for a mobile subagent to open, maintain, and close a “connection” with the SNMP agent. Another purpose is for a mobile agent to dynamically register/unregister a MIB group with the SNMP agent. The SNMP agent can also use the control messages to unregister a MIB group, and to close a connection. The “data messages” are used for the SNMP agent to access extended MIB variables implemented in the mobile subagent when a NMS requires to query or set the MIB variables. The “trap message” is used for a mobile subagent to notify the NMS of certain events. When the SNMP agent receives a trap from a mobile subagent, it converts the trap into a SNMP trap and forwards it to the NMS.

Figure 2.3 Mobile Agents Extends SNMP MIB

The requirement for this type of communication is that the legacy SNMP agent should support dynamic MIB extensions requested from the MCE.

2.2.2 Mobile Agents Access the Legacy SNMP Agent MIB

This type of communication is that a mobile agent requires access to the MIB in the legacy SNMP agent. An example of such a mobile agent is a *mobile diagnostic agent* (MDA) which needs information from the SNMP agent. When a MDA arrives at a network element, it first collects related data by querying the local legacy SNMP agent. The local legacy SNMP agent should respond to the queries. After necessary data collection, the MDA processes the data to localize possible faults and causes. If any faults and causes are detected, the MDA may try to “fix” the problem by sending “set” type of requests to the SNMP agent. Figure 2.4 shows the scenarios.:

Figure 2.4 Mobile Diagnostic Agent Accessing SNMP MIB

A variation of this scenario is that the MDA can also transparently access extended MIB groups implemented in a mobile subagent as depicted in Figure 2.5. In this case when received a request for a MIB variable from the MDA, the SNMP agent will send the request to the MSA if the variable is implemented in the subagent. The SNMP agent will respond the MDA request when it receives the response from the MSA.

This type of communication is similar to the way that a NMS accesses SNMP agents. The major difference is that the communications between a mobile diagnostic agent and a SNMP agent are in the same network element while the communications between a NMS and SNMP agents are across networks. Therefore, a lightweight communication protocol rather than SNMP is preferred.

Figure 2.5 Mobile Diagnostic Agent Accessing Extended SNMP MIB

The protocol for supporting this type of communication can be developed by modifying a subset of the “data access messages” in the DPI protocol. This small subset of messages includes *get*, *getnext*, *set*, and *response*. They should be used in reverse, which means that, instead of SNMP agent pulling mobile subagents, other mobile agents pull the SNMP agent. Therefore, this protocol can be called a Reversed DPI protocol (RDPI).

In summary, the requirement for this type of communication is that the legacy SNMP agent should support a lightweight protocol such as RDPI for mobile agents to access its static and extended MIB objects.

2.2.3 Mobile Agents Both Extend and Access the Legacy SNMP Agent MIB

This type of communication is a combination of the above two, that is, a mobile agent not only extends but also accesses the MIB in the SNMP agent. This type of communication can be better explained by using the *meta-variable* concept. A *meta-variable* is a MIB variable that only exists in a MIB definition document and is a function of real MIB variables. MIB designers use *meta-variables* to express useful derivations that can be made from raw data[17]. For example, an average value of a type of performance data can be derived by first collecting all data followed by a sum and a division operation.

Figure 2.6 Mobile Agents Extend and Access SNMP MIB

At the time the *meta-variable* concept is proposed, it expects that the management station would have to perform the function. This implies that the function must be

expressed by some procedural statement or script that can be mapped down to basic SNMP *get*, *getnext*, and *set* primitives[17]. If the quantity of the raw data is huge, there will be a substantial data transmission between the NMS and the SNMP agent. This is not desirable.

Mobile agents can be used to implement *meta-variables* much more “cheaply”. As is shown in Figure 2.6, the mobile agent that implements *meta-variables* can be called **mobile meta-variable-agent (MMVA)**. The MMVA is registered as a subagent of the SNMP agent. When the SNMP agent receives a query for a *meta-variable* from a NMS, it makes a SNMP message conversion and relays the message to the MMVA. Because the required variable is a function of other MIB variables, the MMVA sends query requests to the SNMP agent to collect the values of those related real MIB variables. After all input data is collected, the MMVA computes the value of the *meta-variable* according to a predefined function with the collected data as the inputs. When the value of the *meta-variable* is obtained, the MMVA forms a response packet and send it back to the SNMP agent. The SNMP agent converts the response message and relays it to the NMS.

Figure 2.7 shows a variation of this type of communication where real variables are implemented in multiple mobile subagents. When the SNMP agent receives a request for a MIB variable which is implemented in the MMVA, it makes a protocol conversion, sends the message to the MMVA, and waits for the response. When the MMVA receives the query for a *meta-variable*, it first builds a request packet with all the input variables of the *meta-variable* function as the packet’s variable-binding list, and then sends it back to the SNMP agent, and waits for the response. When the SNMP agent receives a request from the MMVA, it searches the MIB variables. If the MIB variables are found in extended MIB groups of mobile subagents (MSAs), the SNMP agent will create request packets, send them to the corresponding mobile subagents, and then wait for the responses. After receiving all the responses, the SNMP agent will build a response packet using the values from all mobile subagents’ response packets, and then send it back to the MMVA. When

the MMVA receives the response packet, it uses the variable values of the response packet as its input data and calculates the *meta-variable* value according to a pre-defined function. After the calculation, the MMVA creates a response packet and send it back to the SNMP agent. The SNMP agent will make a protocol conversion and send it back to the NMS.

Figure 2.7 MMVA Access Extended SNMP MIB

The requirement for this type of communication is that the legacy SNMP agent should not only support dynamic MIB extension, but also support data query from mobile agents.

2.3 XMS-SNMP Agent Functional Architecture

Based on the above requirements, a functional architecture for the legacy SNMP agent to support the MCE is depicted in Figure 2.8. The legacy SNMP agent is modified into an eXtensible Mobile Supported (XMS) SNMP agent.

Figure 2.8 XMS-SNMP Agent Functional Architecture

As is shown, the XMS-SNMP agent has one MIB engine, one static MIB and three interfaces, including SNMP, DPI, and RDPI interfaces. The SNMP interface supports

conventional manager and agent communications using standard SNMP protocol. There is no change here.

The DPI interface is used to support the first type of communication between the MCE and the XMS-SNMP agent. A mobile subagent can send an *open* message to the agent requesting a connection. After the connection is granted by the XMS-SNMP agent, the mobile subagent can send a “register” message to the agent, requesting registration of a MIB group. When all the registrations have been completed, the mobile subagent waits for requests from the XMS-SNMP agent for access to extended MIB objects. Valid requests from the agent are *get*, *getnext*, *set*, *commit*, *action*, and *undo*. Both agent and the subagents may unregister a MIB group or close the connection at any time.

The RDPI interface is used to support the second type of communication between the MCE and the SNMP agent. As with the SNMP protocol, a mobile RDPI agent can send *get*, *getnext*, and *set* requests to the SNMP agent. The difference between the RDPI and SNMP is that the RDPI is a more lightweight protocol. The two are compared in Appendix A.

The DPI and RDPI protocols can be used as a pair by a mobile agent to implement the third type of communication.

The MIB engine is the core of the XMS-SNMP agent. It is used to dispatch various requests from different interfaces and send responses back.

The static MIB has two purposes. One of them is to support original MIB implementations such as MIB-II. Another is to manage those dynamically-registered mobile agents and extended MIB groups. The static MIB not only maintains various information about the mobile agents, but also acts as a vehicle to clean orphaned objects when the connections between the XMS-SNMP agent and the subagents are broken. Full descriptions of this function are provided in Chapter 5.

CHAPTER 3

INTEGRATION PROTOCOLS

The functional architecture of the XMS-SNMP agent defined in Chapter 2 requires a DPI and a RDPI protocol to support various requests from the Mobile Code Environment (MCE). This chapter specifies the RDPI protocol and describes the enhancement of the DPI protocol to meet the communication requirements. The DPI protocol specification can be found in [18] and is not repeated here.

3.1 RDPI Protocol

The purpose of the RDPI protocol is to enable a mobile agent to access MIB objects in the XMS-SNMP agent. It includes four types of packets: *get*, *getnext*, *set*, and *response*. All packets share a common header. The header and each packet are described in the following subsections respectively.

3.1.1 RDPI Header

Table 3.1 RDPI Packet Header

Offset (octet)	Field
0	packet length to follow (MSB to LSB)
2	protocol major version (=2)
3	protocol minor version (=2)
4	protocol release (=0)
5	packet ID (MSB to LSB)
7	packet type

The RDPI packet header is tabulated in Table 3.1. It is the same as the DPI packet header. Every RDPI packet includes this header. The packet length, which is the first two octets, indicates the number of octets to follow. The protocol major version is intended to

indicate, in broad terms, what version of the protocol is used. The protocol minor version is intended to identify major incompatible versions of the protocol. The protocol release is intended to indicate incremental modifications to the protocol. The next field is the packet ID which contains a packet's identification. Mobile agents and the XMS-SNMP Agent can use this field to match responses with requests. The packet ID is a monotonically increasing unsigned 16-bit integer which wraps at its maximum value. The last field indicates the type of the packet to be processed. Its values of 1, 2, 3, 5 correspond respectively to *get*, *getnext*, *set*, and *response*.

In this implementation, the protocol information, including major version, minor version, and release number, use the same values as defined in DPIv2[18]. The purpose is to share the same API (implemented in C) of the DPI protocol.

3.1.2 GET/GETNEXT

Table 3.2 RDPI GET/GETNEXT Packet

Offset (octet)	Field
0	packet length to follow (MSB to LSB)
2	protocol version (=2)
3	protocol minor version (=2)
4	protocol release (=0)
5	packet id (MSB to LSB)
7	packet type = GET/GETNEXT (=1/2)
8	community name length (MSB to LSB)
10	community name (if any)
10+L1	object ID with two nulls
10+L2	optionally more variable-bindings (object ID)
notes	L1 = length of community name L2 = L1 + strlen(object id) + 2

The *get* and *getnext* are used for a mobile agent to query MIB objects in an XMS-SNMP agent. Table 3.2 shows their packet layout. The *get* packet queries the exact objects specified by the object IDs, while the *getnext* packet queries the objects next to the specified object IDs. The community name, as with the SNMP message, is used to as a password for the *get/getnext* operation. The community name length indicates the number of octets in the community name. The object ID defines the variable to be queried. Optionally, there may exist more than one object ID for each packet.

3.1.3 SET

Table 3.3 RDPI SET Packet

Offset (octet)	Field
0	packet length to follow (MSB to LSB)
2	protocol version (=2)
3	protocol minor version (=2)
4	protocol release (=0)
5	packet id (MSB to LSB)
7	packet type = SNMP_DPI_SET (=3)
8	community name length (MSB to LSB)
10	community name (if any)
10+L1	object ID with two NULLs
10+L2	Variable Type
10+L2+1	Length of Value (2 octets, MSB to LSB)
10+L2+3	Value
10+L3	optionally more variable-bindings (object id, type, length, value)
notes	L1 = length of community name L2 = L1 + strlen(object id) + 2 L3 = L2 + 3 + length of value

This packet is intended for a mobile agent to change one or more MIB variables in an XMS-SNMP agent. Table 3.3 shows the packet layout. The role of the community name

and community name length is the same as that for the *get/getnext* packet. Each variable-binding includes four fields: an object ID, an object type, the length of the value, and the value. Optionally, a *set* request can have more than one variable-binding.

3.1.4 RESPONSE

Table 3.4 RDPI RESPONSE Packet

Offset (octet)	Field
0	packet length to follow (MSB to LSB)
2	protocol version (=2)
3	protocol minor version (=2)
4	protocol release (=0)
5	packet id (MSB to LSB)
7	packet type = SNMP_DPI_RESPONSE (=5)
8	error code (1 octet)
9	error index (4 octets, MSB to LSB)
13	object id with two NULLs
13+L1	SNMP Variable Type
13+L1+1	Length of Value (2 octets, MSB to LSB)
13+L1+3	Value
13+L2	optionally more variable-bindings (object id, type, length, value)
notes	$L1 = \text{strlen}(\text{object id}) + 2$ $L2 = L1 + \text{length of value} + 3$

This packet is used for an XMS-SNMP agent to reply to a request of *get*, *getnext*, or *set*. The layout of this packet is tabulated in Table 3.4. If a request fails to access a variable, the error code field indicates the failure reason, and the error index indicates which variable-binding has failed.

As can be seen, the RDPI messages are a small subset of the “DPI protocol

messages”. The messages only include *get*, *getnext*, *set*, and *response*. The *get* and *getnext* messages are used for a mobile agent to query MIB variables in the XMS-SNMP agent, while the *set* message is used for a mobile agent to change MIB variables in the XMS-SNMP agent. For each request message, the XMS-SNMP agent uses a *response* message to reply.

The RDPI messages are modified from the corresponding DPI messages. The major modification is that, instead of having a *group ID* and an *instance ID* for each variable in a DPI message, a RDPI message has only one *object ID* for each variable since the applications of the RDPI protocol do not have the concept of *group ID* and *instance ID*. In DPI messages, the *group ID* and the *instance ID* are two strings terminated by a NULL character. In order to be compatible with the DPI’s application program interface, the object ID in an RDPI message is terminated by two NULLs.

3.2 DPI Protocol Enhancement

The DPI protocol is used for a mobile subagent to dynamically extend MIB groups to the XMS-SNMP agent. The extended MIB variables can be accessed transparently by NMS using SNMP protocol. Internally, the XMS-SNMP agent has to perform protocol conversions between the SNMP and DPI. For a SNMP *set* request, there is a mismatch problem between the two protocols. To solve the problem, an addition of the *action* packet type is required. This section briefly discusses the rationale for such an addition. Appendix B describes the test scenarios which uncovered the problem.

3.2.1 UCD-SNMP Agent SET Operations

For a SNMP set request from a network management system (NMS), the UCD-SNMP[22] agent goes through four phases: *RESERVE1*, *RESERVE2*, *COMMIT*, and *ACTION* phases. This is to achieve the effect of “simultaneously” changing multiple

variables per *set* request.

In the first phase, the agent issues a “*RESERVE1*” operation to each variable in the list. This is intended to give every variable an opportunity to “reserve” some “resources”. If any one of the variables fails to allocate its required resources, the agent requests each variable to free its allocated resources by issuing a “*FREE*” operation, and stops further processing.

In the second phase, the agent issues a *RESERVE2* operation to each variable in the list. This is to give each variable an opportunity to do a “pre-set”. If any one of the variables fails to do the “pre-set”, the agent requests each variable to “undo” the “pre-set” by issuing a “*FREE*” operation, and stops further processing.

In the third phase, the agent issues a *COMMIT* operation to each variable in the list. If any one of the variables fails to perform the *COMMIT*, the agent requests each variable to “undo” the change by issuing a “*FREE*” operation, and stops further processing.

In the fourth phase, the agent issues an *ACTION* operation to each variable. This is to give each variable an opportunity to perform any pre-defined actions against the value change.

It should be pointed out that in each phase, the operation is preceded by a *get* operation.

3.2.2 DPI *SET* Operations

For a *set* type of operation, the DPI protocol only defines three types of request messages, including *set*, *commit*, and *undo*. These three messages can be used in the following three combinations:

SET, COMMIT

SET, UDNO

SET, COMMIT, UNDO

As with the *set* type of operation in the SNMP agent, these combinations can be

used in two phases: *set* and *commit*. In the first phase, a DPI subagent requests each variable in the variable-binding list to perform a *set* operation. If any of the variables fails to perform the operation, the subagent requests each variable to perform an “undo” operation, and stops further processing.

In the second phase, the DPI subagent requests each variable to perform a *commit* operation. If any variable fails to perform the operation, the subagent requests each variable to perform an “undo” operation, and stops further processing.

3.2.3 Matching DPI SET Operations with SNMP SET Operations

Obviously, the two phases of the DPI *set* type operations do not match the four phases of the SNMP *set* operations. To match the DPI *set* operations with the UCD-SNMP *set* operations, at least an “ACTION” type of message needs to be added to the DPI protocol. Otherwise, the communication between the UCD-SNMP agent and the DPI subagents will not work properly in some situations (for further description, see Appendix B). After the addition of the “ACTION” type message to the DPI protocol, the mappings from the UCD-SNMP *set* operations to the DPI *set* operations are illustrated in the following table:

Table 3.5: Mapping SNMP SET Operations to DPI SET Operations

SNMP SET OPERATIONS	MAPPING	DPI SET OPERATION
RESERVE1	----->	SET
RESERVE2		
COMMIT	----->	COMMIT
ACTION	----->	ACTION
FREE	----->	UNDO

The SNMP *SET RESERVE2* does not have a correspondence in the DPI *SET* operation. This is permissible as long as the DPI *SET* operation performs the functions equivalent to the *RESERVE1* and *RESERVE2* functions in the SNMP *SET* operation.

CHAPTER 4

XMS-SNMP AGENT DESIGN

This chapter describes the XMS-SNMP agent design which implements the functional architecture defined in Chapter 2. Since the design is based on the public CMU and UCD-SNMP agent [22], section 4.1 analyzes the pertinent data models of two agent implementations. Section 4.2 presents the modified data model for the XMS-SNMP agent. Section 4.3 describes how the XMS-SNMP agent schedules the processing of different messages from different interfaces (SNMP, DPI, RDPI). In collaboration with the data model, sections 4.4, 4.5, and 4.6 describe the agent's dynamical behavior of processing different types of SNMP, DPI, and RDPI requests respectively.

4.1 CMU-SNMP and UCD-SNMP Agent Implementations

4.1.1 CMU-SNMP Agent Data Model

The fundamental data model of the CMU-SNMPv2 agent implementation is depicted in Figure 4.1. Basically, the data model is a two-dimensional structure. The column on the left represents a table of “struct subtree” entries. The major attributes of the “*struct subtree*”, as shown in Figure 4.2, include a *name* and a pointer, *variables*. The *name* is used to specify a *subtree*'s ID. The *variables* points to an array of “struct variable” elements. The *name_len* indicates the number of sub-OIDs in the *name* attribute. The *variable_len* indicates the number of “struct variable” elements in the array *variables*, and the *variable_width* indicates the size of “struct variable” in bytes.

Figure 4.1 CMU-SNMP Agent Data Model

The attributes of the “struct variable” are shown in Figure 4.3. These attributes will be explained in the following paragraphs with examples.

```

struct subtree {
    oid          name[16];
    u_char       nameLen;
    struct variable* variables;
    int          variables_len;
    int          variables_width;
};

```

Figure 4.2 CMU-SNMP "subtree" Structure

```

struct variable {
    u_char       magic;
    char         type;
    u_short      acl;          /* access control list for variable */
    u_char       *(*findVar)();
    u_char       nameLen;
    oid          name[32];
};

```

Figure 4.3 CMU-SNMP "variable" Structure

This data model is used to organize SNMP MIB objects. Each MIB object is identified by an "object identifier (OID)". An object identifier usually comprises a subtree-ID and a variable-ID¹. For example, *sysDescr* is a MIB variable in the RFC1213 *system* group. The subtree-ID for *sysDescr* is 1.3.6.1.2.1.1, or *iso.org.dod.internet.mgmt.mib-2.system* in a descriptive form. The variable-ID for *sysDescr* is 1. Hence, the object identifier for *sysDescr* is 1.3.6.1.2.1.1.1. The *name* in the "struct subtree" is used to define

1. A subtree-ID is also called a group-ID, and a variable-ID is also called an instance-ID.

an object's subtree-ID, and the *name* in the “struct variable” is used to specify an object's variable-ID.

Each instance of the “struct variable” has a pointer, (**findVar*)(), which refers to an application-specific MIB access routine. This routine function returns this variable's value, write-method, and object identifier.

Usually, MIB variables of a MIB group share one access routine. For example, variables such as *sysDesc* and *sysUptime* in the RFC1213 *system* group share the same access routine named *var_system*() to retrieve their values. To identify which variable a call is for, the “struct variable” defines a *magic* number. The function call uses the *magic* number to further qualify the object identifier. For example, in the *system* group, the *magic* number of each variable is equal to the variable's instance ID.

```
u_char* findVar (struct variable *vp,
                 oid *name,
                 int *length,
                 int exact,
                 int *var_len,
                 int write_method).
```

Figure 4.4 CMU-SNMP MIB Access Function Interface Definition

The SNMP MIB access function interface definition is shown in Figure 4.4. The first parameter, *vp*, is an input parameter which points to an instance of “struct variable”. Information about the variable, such as *magic* number, variable *type*, access control list (*acl*), and variable-ID (*name*) can be obtained through this pointer. The variable *type* indicates the variable's ASN.1 type, such as INTEGER, DISPLAY STRING. The access control list (*acl*) indicates the variable's access privileges such as whether it is READ_ONLY or READ_WRITE.

The next two parameters, *name* and *length*, are used as both input and output parameters. As input parameters, they provide the object ID for the search and the length of the object ID respectively. As output parameters, these two variables are updated with the object ID and the length of the object ID found by the access function. The input values and output values of these two parameters are different for SNMP *getnext* operations, but are the same for SNMP *get* and *set* operations.

The fourth parameter, *exact*, is used as an input parameter. When its value is *true*, the function performs a *get* operation, and when it is *false*, the function performs a *getnext* operation.

The fifth parameter, *var_len*, is an output parameter. When the function call is returned, the searched value is returned as an array of “*unsigned char*”. The size of the array is updated in *var_len*.

The last parameter, *write_method*, is an output parameter. When the function call is returned, this variable is updated with a write-method of this variable if the variable is read-writable. This variable is updated with zero if the variable is read-only.

It takes a few steps to obtain the value of a variable. For example, when the agent receives a SNMP *get* request, after parsing and extracting the object ID from the *get* packet, it first searches the *subtrees* table to find the correct *subtree* by matching the object ID with the subtree-ID specified in the *name* field of each instance of “struct subtree”. Secondly, it searches the *variables* of the *subtree* to find the variable element by matching the rest of the object ID with the variable-ID specified in the *name* field of each *variable*. Finally, it calls the access function pointed by the *findVar* to return the variable's value and other related parameters such as *write_method*. The *write_method* is used for SNMP *set* requests.

As is shown in Figure 4.1, the *subtrees* is a table of “struct subtree” entries. Therefore, the size of the *subtrees* is fixed. It is impossible to add a new MIB *subtree* to the agent's *subtrees* without modifying and re-compiling the agent code.

4.1.2 UCD-SNMP-V3.1 Data Model

Basically, the UCD-SNMP-V3.1 agent data model is the same as the CMU-SNMPv2 agent data model except that, instead of being statically defined as a table of “struct subtree” entries (struct subtree subtrees[]), the *subtrees* is defined as a pointer to an array of “subtree” entries (struct subtree* subtrees). This change is for the support of configuring extensible MIB groups.

The UCD-SNMP agent has defined several additional MIB groups, such as *processes*, *exec*, *disk*, *memory group*, and so on [22]. Instances of MIB variables in these MIB groups can be added to the agent through the configuration file *snmpd.conf*. For example, for the *processes* MIB group, when users want to use the agent to monitor if a certain process, e.g. *mountd*, is running on the host, an entry such as “*proc mountd*” can be added to the configuration file. Thereafter, whenever the agent detects that there is no *mountd* process running, the agent will raise an error flag.

When the SNMP daemon is started, it automatically reads the configuration file, calculates the size of the extended MIB group, and dynamically allocates a chunk of memory space for both the built-in MIB groups and the extended MIB groups. The extended MIB groups and the built-in MIB groups are then re-built into the *subtrees* in the lexicographical order. Old memory space are released.

The extensible MIB groups configuraiton can also be done after the SNMP daemon is started. This is achieved by first modifying the configuration file, then forcing the SNMP daemon to re-read the configuration file. After reading the configuration file, the agent re-calculates the size of the re-configured MIB groups and allocates appropriate memory space. The built-in and the re-configured MIB groups are re-built into the *subtrees* in the lexicographical order. The old allocated memory spaces are released,

The way to force the SNMP daemon to re-read the configuration file is to set a variable called *updateConfig* in the *version group* to “1”. [22]

In summary, the UCD-SNMP-V3.1 agent has the following advantages over the CMU-SNMP agent:

- Memory for MIB groups is dynamically allocated.
- It is suitable for batch MIB group updating via the configuration file.

However, the UCD-SNMP-V3.1 agent still has the following limitations:

- Only the instances of the MIB variables in the defined MIB groups can be configured. New MIB groups cannot be added at run-time.
- Not suitable for updating MIB *group* on a one-on-one base.

4.2 XMS-SNMP Agent Data Model

The XMS-SNMP agent data model is depicted in Figure 4.5. The top of the figure is a list of *subtrees* pointed by the *subtreeList*. Each *subtree* is either statically or dynamically registered. A built-in *subtree* has a pointer, *variables*, which refers to an array of variables. A dynamic *subtree* has the pointer, *variables*, pointed to a *registerTreeList*.

The register trees in a *registerTreeList* are arranged in the order of priority. A register tree with a higher registration priority precedes a register tree with a lower registration priority. Each register tree in the list has two pointers, one pointing to its *subtree*, and the other pointing to its *subagent*.

The bottom of the figure is a list of *subagents* pointed by the *subagentList*. Each *subagent* has a *regTreeSet* which is an array of pointers to the *registerTrees* the subagent has registered.

The main idea behind the picture is that one mobile subagent can support multiple MIB groups, and one MIB group can be registered by multiple mobile subagents.

Figure 4.5 XMS-SNMP Agent Data Model

4.2.1 “subtree” Data Structure

Figure 4.6 shows the “subtree” data structure. It is made by adding an extra attribute, *link*, to the corresponding data structure in the CUM/UCD implementation. The type of the attribute *link* is “struct doubleLink” which is also defined in the figure. The “struct doubleLink” defines two pointers, *prev* and *next*, which are used to point its preceding *subtree* and next *subtree* respectively. The purpose of defining such a data structure is for fast adding/deleting *subtrees* to/from the *subtree* list at run-time.

```

struct subtree {
    doubleLink      link;
    oid             name[16];
    u_char          nameLen;
    struct variable* variables;
    int             variables_len;
    int             variables_width;
};

where

struct doubleLink{
    struct doubleLink* prev;
    struct doubleLink* next;
};

```

Figure 4.6 XMS-SNMP Agent “subtree” Structure

The “*name*” in the “struct subtree” is used to specify a *subtree*’s object identifier, and the “*nameLen*” indicates the number of sub-ids in the *name*.

The *variables* in the “struct subtree” points to an array of “struct variable” instances. The “struct variable” is the same as that defined in CMU/UCD code. The

variable_len is the number of “struct variable” instances in the above array, and the *variable_width* is the size of the “struct variable” in bytes.

For a dynamically registered *subtree*, the *variable_length* is set to 0, and the *variables* is cast into a pointer to a *registerTreeList* which is shown in Figure 4.5.

4.2.2 “registerTree” Data Structure

Figure 4.7 shows the “registerTree” data structure. The first attribute, *link*, is used to point to a *registerTree*’s preceeding *registerTree* and succeeding *registerTree* within the scope of a *subtree*. The second attribute, *subtreePtr*, is used to point to the *subtree* to which the *registerTree* has registered with. Attribute *subagent_p* is used to refer to the *subagent* which has registered this *registerTree*. Figure 4.5 shows this relationship graphically.

```

struct registerTree {
    struct doubleLink    link;
    struct subtree*      subtreePtr;
    u_short              timeout;
    long                 priority;
    char                 bulkSelect;
    char                 viewSelect;
    struct subagent *     subagent_p;
    int                  adminStatus;
};

```

Figure 4.7 XMS-SNMP Agent *registerTree* Structure

Attribute *adminStatus* is solely used for maintenance purpose which will be discussed in detail in Chapter 5.

The rest of the attributes, including *timeout*, *priority*, *bulkSelect*, and *viewSelect* are

extracted from a *register* request packet sent from a mobile subagent. The *timeout* value is the requested timeout from a subagent. The agent will use this value (or its maximum) to await a response to requests for this registered *subtree*.

The value of the *priority* is the priority assigned to the subagent by the agent. In the case where a *subtree* is registered by multiple subagents, MIB variable access is given to the subagent with the highest priority.

The *bulkSelect* is an indication as to whether the subagent wishes to handle the *getbulk* itself. If not, then the agent will translate *getbulk* into multiple *getnext* requests. This feature is not supported in this implementation.

The *viewSelect* is an indication as to whether the subagent wishes to handle MIB view selection (SNMPv1 community string authentication) for DPI *get*, *getnext* or *set*, *commit*, *undo*, *action* requests from the agent.

4.2.3 “subagent” Data Structure

The “subagent” data structure is defined in Figure 4.8. The first attribute, *link*, is used to link all *subagents* together. The second attribute, *regTreeSet*, is used to refer to the *registerTrees* this subagent has registered. The bottom of Figure 4.5 shows this relationship.

Attribute *adminStatus* is used for maintenance purpose which is described in detail in Chapter 5.

Attribute *openTime* is the time when the agent received a DPI *open* request from a subagent. It is used to indicate the period of time since the subagent has “opened” the connection. The value of this attribute is equal to the *sysUpTime* at that moment.

The rest of the attributes are extracted from the DPI *open* packet. Attribute *address* is used to identify the remote subagent. It contains an IP address and a port number in the case of UDP.

Attribute *maxVarBinds* is used to indicate the maximum number of variable-bindings per DPI packet that this subagent is prepared to handle. Since the XMS-SNMP

agent sends only one variable-binding a time to a subagent, this variable has been used to indicate the maximum number of MIB groups or sub-trees a subagent to register in the "open" request packet.

```

struct subagent {
    struct doubleLink    link;
    struct regTree**     regTreeSet;
    struct sockaddr_in    address;
    u_long               openTime;
    oid*                 oid_p;
    int                  oid_len;
    char*                description_p;
    u_long               defaultTimeout;
    int                  max_varBinds;
    char                 character_set;
    u_long               password_len;
    u_char*              password_p;
    int                  adminStatus;
};

```

Figure 4.8 XMS-SNMP Agent "subagent" Structure

The "*oid_p*" points to a subagent OID, which is a unique ASN.1 object identifier. The *oid_len* is the length of the subagent OID. The *description_p* is a pointer to the descriptive name of the subagent.

The *defaultTimeout* is a default timeout value to be used by the XMS-SNMP agent for all requests for objects implemented in this subagent. The timeout value in the *registerTree* overwrites this value[18].

Attributes *character-set* indicates the choice of character set of the subagent. In this implementation, only ASCII character set is considered.

Attributes *password_p* and *password_len* are optional. The agent does not require a subagent to provide the password for an *open* request in this implementation.

4.3 XMS-SNMP Agent Message Scheduling

This section describes the modifications to the UCD-SNMP agent so that messages from DPI and RDPI applications can be scheduled and processed as well.

4.3.1 Adding DPI Port and RDPI Port to the UCD-SNMP Agent

When the CMU-SNMP/UCD-SNMP agent daemon is started, it creates a socket for each port defined in the party configuration file. The created sockets are saved in variable *sdlist[]*, and the number of socket are saved in variable *sdlen*. The definition of these two variables are defined in *snmpd.c* file as is shown in Figure 4.9.

```
int sdlist[32];
int sdlen;
```

Figure 4.9 socket list and socket number variable definitions in *snmpd.c* file

The DPI and RDPI ports are added to the agent in the similar way. The DPI port number is implemented to be equal to the last SNMP port number plus 3. The RDPI port number is equal to DPI port number plus 1. Immediately after the initialization of the SNMP ports, we create a socket for the DPI port, and a socket for the RDPI port, and append the two sockets to the variable *sdlist*. The modification is made in the main function *main()* in *snmpd.c* file.

4.3.2 Scheduling of SNMP, DPI and RDPI Messages

The CMU/UCD-SNMP agent receives and processes SNMP messages from different ports by calling *receive(int sdlist[], int sdlen)* function. The *receive()* function is defined in *snmpd.c* file. The ports correspond to the sockets in parameter *sdlist*; and the

number of ports corresponds to the number of sockets represented by parameter *sdlen*.

In the XMS-SNMP agent, the *receive()* function has been modified to receive and schedule DPI and RDPI messages as well as SNMP messages. The modifications are highlighted in bold face as depicted in Figure 4.10.

```

receive(sdlist, sdlen)
    int sdlist[];
    int sdlen;
{
    .....
    while(1){
        .....
        count = select(numfds, &fdset, 0, 0, tvp);
        if( count > 0 ) {
            for(index = 0; index < sdlen; index++){
                if(FD_ISSET(sdlist[index], &fdset)){
                    if(is_DPI_port(sdlist[index])){
                        dpi_read_packet(sdlist[index]);
                    }else if(is_RDPI_port(sdlist[index])){
                        rdpi_read_packet(sdlist[index]);
                    }else{
                        snmp_read_packet(sdlist[index]);
                    }
                    FD_CLR(sdlist[index], &fdset);
                }
            }
            .....
        }
        .....
    }
    .....
}

```

Figure 4.10 XMS-SNMP Agent Message Scheduling

The function call *select()* is used to check if any of the file descriptors in parameter *fdset* is ready for reading. The upper bound of the file descriptors is defined in the first parameter, *numfds*. The maximum interval to wait for the selection to complete is specified by the last parameter, *tvp*. The third and fourth parameters, which specify the bit masks for *writefds* and *exceptfds* respectively, are equal to zeros and are not of our interests. This

function returns the number of descriptors contained in the bit masks or -1 if an error occurred. If the time limit expires, `select()` returns 0.

If the return from function `select()` is greater than zero, the `receive()` function searches `sdlist[]` to determine which socket has a message by calling `FD_ISSET(sdlist[index], &fdset)`.

If a socket, for example, `sdlist[index]`, has been determined to have a message ready for reading, the agent will call a function call, corresponding to the type of port, to read from the socket. If the socket, `sdlist[index]`, corresponds to a DPI port, it will call `dpi_read_packet(sdlist[index])`. If the socket, `sdlist[index]`, corresponds to a RDPI port, it will call `rdpi_read_packet(sdlist[index])`. Otherwise, the agent will call `snmp_read_packet(sdlist[index])` which corresponds to a SNMP port.

To verify if a socket, `sd`, corresponds to a DPI port, the system routine, `getsockname(sd, (struct sockaddr *)&sockName, &sockNameLen)` can be called to obtain the `sockName`. The port number is the attribute of `sockName.sin_port`. If this port number is equal to the DPI port number, the socket, `sd`, is said to correspond to the DPI port. Similarly, a socket can be verified if it corresponds to a RDPI port.

4.4 SNMP Request Processing

When the `snmp_read_packet(int sd)` routine is called by the agent, it will perform the following functions:

- read the message by calling system routine `recvfrom()`;
- parse and process the message by calling `snmp_agent_parse()`;
- reply the request by calling system routine `sendto()`;

The `snmp_agent_parse()` routine is used to parse and process SNMP messages. In order to access MIB variables in a DPI subagent, the `snmp_agent_parse()` will call the following routines indirectly: `getStatPtr()`, `findVarInRegSubtree()` and

writeVarInRegSubtree(). These three routines are described in the following subsections.

4.4.1. *getStatPtr()* Routine

This routine can be better understood with the help of Figure 4.5, and Figure 4.3. For each variable in the variable-binding list, the *getStatPtr()* routine is called to search the *subtreeList* to find the *subtree* by matching the requested object identifier with the subtree-ID of each *subtree*. If the *subtree* located is a built-in *subtree* (*variables_len* is not 0), it continues to search the *variables of the subtree* to find the *variable* by using the rest of the OID of the requested variable to match the *variable-ID* of each *variable*. It accesses the requested variable by calling the *findVar()* function defined in the located *variable*.

If the *subtree* located is a dynamically registered *subtree* (*variables_len* is 0), the *getStatPtr()* function will call *findVarInRegSubtree()* routine to return the variable's value, and write method. This routine is described in the next subsection.

4.4.2. *findVarInRegSubtree()* Routine

This routine is designed to perform the following functions when it is called:

- create a DPI *get* packet for SNMP *get* request or a DPI *getnext* packet for SNMP *getnext* request.
- send the DPI packet to the subagent. The *subtree's variables* points to its *registerTreeList*. The first *registerTree*, which has the highest *priority*, in the *registerTreeList* has a pointer to the subagent.
- wait for the response from the subagent. The time-out value set for waiting the response is obtained from the *registerTree's timeOut* attribute, or from the subagent's *timeOut* attribute in case that the *registerTree's timeOut* attribute is not defined.

- If it does receive the response before the *timeOut* value expires, it will parse the response packet, and return the value, write method, and other parameters. The write method returned is *writeVavInRegSubtree()*.
- If it could not receive the “response” within the set period of time, this routine will simply return a NULL.

4.4.3. *writeVarInRegSubtree()* routine

This routine is used to change a MIB variable defined in a DPI subagent. When it is called, it will perform the following functions:

- create a DPI *set*, or *commit*, or *action* packet corresponding to a SNMP *set* “RESERVE1” or “COMMIT”, or “ACTION” respectively.
- send the DPI packet to the subagent that has registered the first *registerTree* in the MIB group’s *registerTreeList*.
- wait for the response from the subagent. The time-out value set for waiting the response is obtained from the *registerTree*’s *timeOut* attribute, or from the subagent’s *timeOut* attribute in case that the *registerTree*’s *timeOut* attribute is not defined.
- parse the response packet.
- return the response.

4.5 DPI Request Processing

When the *dpi_read_packet(int sd)* routine is called by the agent, it will perform the following functions:

- read the message by calling system routine: *recvfrom()*
- parse and process the message by calling: *snmp_dpi_agent_parse()*
- reply by calling system routine: *sendto()*

The *snmp_dpi_agent_parse()* function first parses the incoming binary data and builds a structured DPI packet, and then processes the DPI packet according to the packet type. The following subsections will describe how different types of DPI requests from a DPI subagent are processed.

4.5.1 DPI *open* Request

For a DPI *open* request from a subagent, the agent will search the *subagentList* to verify if such a *subagent* exists by comparing each *subagent's address* in the *subagentList* with the subagent address from the request packet. If such a *subagent* does not exist, the agent will create a *subagent*, fill in the attributes, append the *subagent* at the end of the *subagent* list, and send back a success *response* packet to the subagent. Otherwise, the agent will reject the connection request by sending back a failure *response* packet.

4.5.2 DPI *register* Request

For a DPI *register* request from a subagent, first, the agent will verify if the subagent has “opened” a connection with the agent. The agent will search the *subagentList* and compare the address extracted from the request packet with the *address* of each *subagent*. If the agent fails to find the *subagent*, the agent will send back a *response* with a failure reason “*mustOpenFirst*”, and stop further processing of the request.

Secondly, the agent will verify if the MIB group of the *register* request has been already registered by the subagent. The agent will go over each *registerTree* in the *subagent's regTreeSet* to find its registered *subtree*, and compare the *subtree's* ID with the MIB group ID from the *register* request. If the agent finds a match, that means that the agent has registered this MIB group for the *subagent* before, the agent will reject the request by sending back a failure response with reason “*duplicated registration*”, and stop further processing of the request.

Third, the agent will verify if the requested MIB group is on the *subtreeList*. The agent will search the *subtreeList* and compare each *subtree's* ID with the MIB group ID

from the request packet. There are three possibilities:

- The first possibility is that there is no such a MIB group on the *subtreeList*. In this case, the agent will create a *subtree* and a *registerTree*, copy related information from the request packet, setup appropriate links, and insert them into the *subtreeList* and the *registerTree* respectively.
- The second possibility is that the requested MIB group is on the *subtree* list, and the located *subtree* is a dynamically registered *subtree*. In this case, there is no need to create a *subtree* object. The agent will create only a *registerTree*, copy related information from the request packet, setup appropriate links, and insert it into the *registerTreeList* under this *subtree*.
- The third possibility is that the requested MIB group is on the *subtree* list, but the located *subtree* is a built-in *subtree*. In this case, the agent simply reject the request, since built-in *subtree* can not be shared by subagents.

4.5.3 DPI *unregister* Request

For a DPI *unregister* request from a subagent, first, the agent will find the corresponding *subagent* in the agent's *subagentList*. The agent will search the *subagentList* and compare each *subagent's* *address* with the address extracted from the request packet. If the agent could not find a match, that means that the subagent has no logical connection with the agent, the agent will send the subagent a failure response with a reason *SNMP_ERROR_genErr* and stop further processing of the packet.

Secondly, the agent will find the *registerTree* in the *subagent's* *regTreeSet*. The agent will go over each *registerTree* in the *subagent's* *regTreeSet* to locate the *subtree* and compare the *subtree's* ID with the MIB group ID of the request. If the agent fails to find a match, the agent will send a failure response with a reason *SNMP_ERROR_genErr*.

Finally, the agent will release related resources. It will delete the *regsiterTree* from the *registerTreeList*, nullifies the corresponding entry in the *subagent's* *regTreeSet*. If the

registerTreeList is empty, it will also delete the *subtree* from the *subtreeList*.

4.5.4 DPI *close* Request

When the agent receives a DPI *close* request from a subagent, first, it will locate the *subagent* from the *subagentList* by comparing each *subagent's address* with the address from the request packet. If it fails to find the subagent, the agent simply ignores the request.

Secondly, it will go over the *subagent's regTreeSet* to delete and release each *registerTree* from the corresponding *registerTreeList*, and the *subtree* if the corresponding *registerTreeList* is empty.

Third, it will release the *subagent's regTreeSet*, and delete the *subagent* from the *subagentList*.

4.6 RDPI Request Processing

When function *rdpi_read_packet(int sd)* is scheduled by the agent, it will perform the following functions:

- read the message by calling system routine *recvfrom()*;
- parse and process the message by calling *snmp_rdpi_agent_parse()*;
- reply by calling system routine *sendto()*;

The *snmp_rdpi_agent_parse()* function first parses the incoming binary data to build a structured DPI packet¹, and then calls *do_get()* for processing RDPI *get* and *getnext* requests, and *do_set()* for processing RDPI *set* request. The following two subsections will describe these two functions respectively.

4.6.1 RDPI *get/getnext* Request

When the *do_get()* is called, it will walk through the variable-binding list and perform the following functions for each variable:

- retrieve the variable by calling: *getStatPtr()*;

1. The RDPI packet shares the same structure as the DPI packet.

- build/add the variable and value to the variable-binding list for returning;

After every variable is successfully obtained, the function will build a response packet according to the variable-binding list, and return the packet.

The *getStatPtr()* routine is the same as described in section 4.4.1.

4.6.2 RDPI *set* Request

When the *do_set()* function is called, it will perform the similar functions as the SNMP set request does, that is, it will call the function *set_var_bind_list()* four times in the following order:

```
errstat=set_var_bind_list(pack_p, RESERVE1, &index);
errstat=set_var_bind_list(pack_p, RESERVE2, &index);
errstat=set_var_bind_list(pack_p, COMMIT, &index);
errstat=set_var_bind_list(pack_p, ACTION, &index);
```

The first parameter, *pack_p*, is a pointer that points to a “set” type of variable binding list. The second parameter, whose value can be any of *RESERVE1*, *RESERVE2*, *COMMIT*, and *ACTION*, is an action which will be passed to the variable’s access function. The third parameter, *index*, is used to return the index number of the variable which has been failed to access, in the variable-binding list. The returned value which is assigned to *errstat*, is used to indicate if the call succeeds.

If any one of the first three calls fails, it will call the following function to give application a chance to free allocated resources:

```
errstat=set_var_bind_list(pack_p, FREE, &index);
```

The *do_set()* function will make a response and return a response packet at the end of processing. The response packet is either a success response if all the first four function calls succeed or a failure response if any one of the calls fails.

The *set_var_bind_list()* function walks through the variable-binding list and

perform the following functions for each variable:

- retrieve the variable and write method by calling: *getStatPtr()*;
- call the returned write method to perform an “action” against the variable. The “action” value is passed in the order of {*RESERVE1*, *RESERVE2*, *COMMIT*, *ACTION*};

In summary, this chapter has analyzed the data model implemented in CMU-SNMP and UCD-SNMP agent. Based on this analysis, a new data model for the XMS-SNMP agent has been designed. The XMS-SNMP agent can process various messages sent from SNMP, DPI and RDPI applications. Dynamic behavior of the data model has been fully analyzed.

CHAPTER 5

MOBILE AGENT MANAGEMENT

This chapter is focused on mobile agent management from the legacy SNMP agent's point of view. Section 5.1 analyzes the possible approaches to manage different types of mobile agents through the XMS-SNMP agent. Section 5.2 defines the structure and contents of the management MIB. The formal management-MIB definition is provided in Appendix C. The management-MIB provides information about mobile subagents and their dynamically registered MIB groups. It also provides a mechanism to clean orphaned objects when the connections between the XMS-SNMP agent and its mobile subagents lost broken. Sections 5.3 and 5.4 describe the management MIB implementation.

5.1 Mobile Agent Management Analysis

As discussed in Chapter 2, there are four types of mobile agents from the XMS-SNMP agent's point of view:

- A mobile agent which registers itself as a mobile subagent with the XMS-SNMP agent
- A mobile agent which only accesses MIB variables in the XMS-SNMP agent or in the subagents of the XMS-SNMP agent
- A mobile agent which performs the functions of the above two
- A mobile agent which does not need to communicate with the XMS-SNMP agent

There are three possible approaches to manage these four types of mobile agents

through the XMS-SNMP agent. The first approach is to directly manage those mobile agents which have registered themselves as subagents to the XMS-SNMP agent.

Information about the subagents and their registered MIB groups can be organized into a static MIB in the XMS-SNMP agent. Therefore, management stations can query the information by sending SNMP messages to the XMS-SNMP agent.

Figure 5.1. Directly Managing Mobile Agents through SNMP

The problem with this approach is that mobile agents other than mobile subagents cannot be controlled through the XMS-SNMP agent. Figure 5.1 shows that the two mobile subagents can be managed through the XMS-SNMP agent, but the MC can not.

The second approach of managing mobile agents is through the Virtually Managed Component (VMC) as described in [1]. This approach is suitable for managing those mobile agents that communicate with the XMS-SNMP agent through the VMC. In this case, the VMC is required to register itself as a subagent to the XMS-SNMP agent.

Information about the mobile agents can be organized into MIB variables and built into the VMC subagent. Therefore, the SNMP-based network management system can access the information of these mobile agents from the VMC subagent.

The problem with this approach is that mobile agents that do not need to communicate with the legacy SNMP agent cannot be managed through the XMS-SNMP agent. Figure 5.2 shows that information about MC1 and MC2 can be accessed from the VMC, but information about MC3 cannot be accessed.

Figure 5.2. Managing Mobile Agents through VMC

The third approach of managing mobile agents is through the Mobile Code Manager (MCM) of the Mobile Code Daemon (MCD). As described in Chapter 2, the principal component of the MCE is the Mobile Code Daemon (MCD). Inside the mobile code daemon, there is a Mobile Code Manager (MCM). The responsibility of the MCM is to maintain these mobile agents. Information such as how many mobile agents in the system, what their characteristics are, and what their states are, can be retrieved from the MCM. Figure 5.3 shows that the MCD is registered as a subagent of the XMS-SNMP

agent. The MCM of the MCD can be registered as extended MIB groups with the agent. This way the MCE and its mobile agents are totally under the control of the XMS-SNMP agent. From the legacy SNMP point of view, the MCD can be called a MCD subagent.

Figure 5.3. Managing MCE through SNMP

These three approaches indicate that a mobile agent can either register itself as a subagent to the XMS-SNMP agent, so that it can be controlled by the XMS-SNMP agent directly, or it can be controlled indirectly through the VMC subagent or MCD subagent. Therefore, controlling subagents of the XMS-SNMP agent is the key to the management. The rest of this chapter will discuss only the subagent management of mobile agents.

5.2 Management-MIB

Figure 5.4 shows the position of the XMS-SNMP agent management-MIB in the

OSI registration tree. It is temporarily under the experimental group. The XMS-SNMP agent management information is organized into two MIB groups: a subagent MIB group and a registerTree MIB group. The following two sub-sections will describe each MIB group in detail respectively.

Figure 5.4. The Position of the XMS-SNMP Agent Management MIB

5.2.1 Subagent MIB Group

The structure of the subagent MIB group is depicted in Figure 5.5. It comprises a *subagentNumber*, and a *subagentTable*. The *subagentNumber* represents the number of subagent entries in the table.

Figure 5.5. "subagentMIB" Structure

Each subagent entry includes a list of read-only attributes about the subagent. The *subagentIndex* is a unique value for identifying a subagent in the table. The *subagentOid* is an ASN.1 object identifier. The *subagentDescription* is a descriptive name of the subagent. The *subagentOpenTime* represents the time the subagent “opens” the connection with the XMS-SNMP agent. Its value is the *sysUpTime* at that point. The *subagentTimeOut* is an

overall default time-out value in seconds. The XMS-SNMP agent will use this value as a time-out value if a register tree does not have its own time-out value. The *subagentIpAddress* and *subagentPortNumber* are the subagent's IP address and port number which uniquely identify the subagent process. The *numbOfRegTrees* is the current number of sub-trees that the subagent has registered. The *maxNumOfRegTrees* is the maximum number of sub-trees a subagent can register. This value is obtained from the maximum number of VarBind in the DPI *open* packet.

Each subagent also has a read-write variable called *subagentAdminStatus*. The purpose of defining this variable is to allow SNMP based management stations to control the subagent. In normal situations, a network management system can close the subagent by setting this variable to 1. In abnormal situations, in which the connection between the XMS-SNMP agent and a subagent is broken, there are orphaned objects left in the XMS-SNMP agent. Management stations can clean these orphaned objects by setting the *subagentAdminStatus* to 1.

5.2.2 Register Tree MIB Group

The structure of the register tree MIB group is depicted in Figure 5.6. It comprise a *registeredTreeTable* only. Each entry of the table has a number of variables describing the attributes of the *registerTree*. The *subagentIndex* and the *registerTreeIndex* are used to uniquely identify an entry in the table. The *registerTreeOid* is the dynamically registered subtree OID.

The *registerTreePriority* is the priority assigned to this *registerTree*. When a dynamic sub-tree has multiple *registerTrees* registered by different subagents, these *registerTrees* are arranged in the order of priorities. A *registerTree* with a higher priority will be placed before the *registerTree* with a lower priority. A request for a variable in the sub-tree from a management station will be sent to the subagent which has the highest priority in the *registerTree*.

Figure 5.6. registerTreeMIB Structure

The *registerTreeTimeOut* is a time-out value in second. When a network manager requests an object in this registered *subtree*, the XMS-SNMP agent will convert and relay the request to the subagent. If the XMS-SNMP agent could not receive the response within the time period defined by the *registerTreeTimeOut*, the XMS-SNMP agent will declare a time-out. If the *registerTreeTimeOut* value is 0, the XMS-SNMP agent will use the default value, that is the *subagentTimeOut*, instead.

The *registerTreeAdminStatus* is a read-write variable. Management stations can unregister a *registerTree* by setting this variable to 1.

5.3 Management-MIB Access Functions

This section discusses how the Management-MIB objects in the MIB definition file can be mapped to the actual MIB variables in the XMS-SNMP agent by the Management-MIB access functions: *var_subagent_number()*, *var_subagent_table*, and *var_registerTree_table*.

5.3.1. Subagent Number Access Function: *var_subagent_Number()*

The MIB object *subagentNumber* is a scalar object and has no correspondent variable in the data model as is shown in Figure 4.5. The access function retrieves *subagentNumber* by counting the number of *subagents* in the *subagentList* in the data model.

5.3.2. Subagent Access Function: *var_subagent_table()*

This sub-section describes how the MIB objects defined in the *subagentTable* as is shown in Figure 5.5 can be retrieved from the XMS-SNMP agent data model as is depicted in Figure 4.5.

The MIB objects in the *subagentTable* are tabular objects. The *subagentIndex* is used to uniquely define a particular row in the table. Each row of the *subagentTable* in the MIB definition file corresponds to a specific *subagent node* in the *subagentList* in the data model. This MIB access function uses the *subagentIndex* value to locate the corresponding *subagent* in the *subagentList* in the data model. As long as the *subagent* is found, the attributes in the *subagentTable* can be readily retrieved from it. For example, attributes *subagentOid* (2), *subagentDescription*(3), *subagentOpenTime*(4), *subagentTimeOut*(5), *maxNumOfRegTrees*(8) and *subagentAdminStatus*(10) can be directly retrieved from the *subagent*'s corresponding variables *oid_p*, *description_p*, *openTime*, *defaultTimeout*, *maxRegTrees*, *adminStatus* respectively. Attributes *subagentIpAddress*(6) and *subagentPortNumber* (7) can be retrieved from the "struct sockaddr_in sAddr". Attribute *numbOfRegTrees*(9) can be retrieved by counting the number of *registerTree* in the

regTreeSet of the *subagent*.

If the *subagentAdminStatus* is set to 1, the access routine will, when it receives an *action* request from the agent, perform the following actions:

- Create and sends a *close* packet to the subagent;
- Remove the *subagent* from the *subagentList*;
- Remove every *registerTree* in the subagent's *regTreeSet* from the corresponding *regTreeList*;
- If any of the *regTreeList* is empty, the access function will also be responsible for removing the corresponding *subtree* from the *subtreeList*;
- Release all those objects.

5.3.3 RegisterTree-MIB Access Function: *var_registerTree_table()*

This sub-section describes how the MIB objects defined in the *registreeTreeTable* as is shown in Figure 5.6 can be retrieved from the XMS-SNMP agent data model as is depicted in Figure 4.5.

As is shown in Figure 5.6, the MIB objects in the *registerTreeTable* are tabular objects. The *subagentIndex* and the *registerTreeIndex* are used to uniquely define a row in the table. Each row of the table corresponds to a particular *registerTree* in one of the *registerTreeLists* as is shown in Figure 4.5. For a request from a management station, the access function of this MIB group will first locate the *subagent* by counting the *subagentIndex* from the *subagentList*. Secondly, the access function will locate the *registerTree* by counting the *registerTreeIndex* from the *subagent's regTreeSet*. As long as the *registerTree* is found, other attributes can be retrieved from the *registerTree* accordingly. Attributes *registerTreePriority*, *registerTreeTimeOut*, and *registerTreeAdminStatus* can be retrieved directly from the *registerTree's priority*, *timeout*,

and *adminStatus* respectively. Attribute *registerTreeOid* can be indirectly retrieved from this *registerTree*'s *subtree* referenced by the *subtreePtr*.

If the *adminStatus* of a *registerTree* is set to 1, the access function will, when it receives an "action" request, perform the following actions:

- Create a DPI unregister packet and send it to the subagent.
- remove the *registerTree* from the *registerTreeList*.
- If the *registerTreeList* is empty, remove the *subtree* as well.
- Nullify the pointer in the *regTreeSet* for this *subagent*.
- free all those objects

5.4 Integration of the Management-MIB Access Functions with Agent

Two steps are required to integrate the Management-MIB access functions into the agent. The first step is to define two tables of variables as are shown in Figure 5.7. One is the subagent variable table: *subagent_variables*. Another is the register tree variable table: *register_tree_variables*. In each table entry, the last field is the instance ID of the variable. The second field from the last is the number of instance sub-ID. The forth field is the access function call, through which the value of the variable identified by the instance ID can be obtained. Other fields are for further qualifying of the access.

```
#define SUBAGENTMANAGMENTMIB      1,3,6,1,3,100
#define SUBAGENTMIB                1,3,6,1,3,100,1
#define REGISTRATIONMIB            1,3,6,1,3,100,2

struct variable4 subagent_variables[] = {
    {SUBAGENTNUMBER,      INTEGER,   RONLY, var_subagent_number, 1, {1}   },
    {SUBAGENTINDEX,       INTEGER,   RONLY, var_subagent_table,  3, {2,1,1}},
    {SUBAGENTOID,         OBJID,     RONLY, var_subagent_table,  3, {2,1,2}},
    {SUBAGENTDESC,        STRING,    RONLY, var_subagent_table,  3, {2,1,3}},
    {SUBAGENTOPENTIME,    TIMETICKS, RONLY, var_subagent_table,  3, {2,1,4}},
    {SUBAGENTTIMEOUT,     INTEGER,   RONLY, var_subagent_table,  3, {2,1,5}},
    {SUBAGENTIPADDRESS,   IPADDRESS, RONLY, var_subagent_table,  3, {2,1,6}},
    {SUBAGENTPORT,        INTEGER,   RONLY, var_subagent_table,  3, {2,1,7}},
    {NUMBEROFGROUPS,      INTEGER,   RONLY, var_subagent_table,  3, {2,1,8}},
    {SUBAGENTMAXREGTREES, INTEGER,   RONLY, var_subagent_table,  3, {2,1,9}},
    {SUBAGENTADMINSTATUS, INTEGER,   RWRITE, var_subagent_table, 3, {2,1,10}},
};

struct variable4 register_tree_variables[] = {
    {REGSUBAGENTINDEX,    INTEGER,   RONLY, var_reg_table, 3, {1,1,1}},
    {REGTREEINDEX,        INTEGER,   RONLY, var_reg_table, 3, {1,1,2}},
    {REGSUBTREEOID,       OBJID,     RONLY, var_reg_table, 3, {1,1,3}},
    {REGPRIORITY,         INTEGER,   RONLY, var_reg_table, 3, {1,1,4}},
    {REGTIMEOUT,          INTEGER,   RONLY, var_reg_table, 3, {1,1,5}},
    {REGADMINSTATUS,      INTEGER,   RWRITE, var_reg_table, 3, {1,1,6}},
};
```

Figure 5.7. Subagent and Register Tree Variables

The second step is to include the *subagent_variables* and the *register_tree_variables* into the built-in *subtree* table, *subtrees*, as is shown in Figure 5.8. The first attribute of each entry is a “struct doubleLink” *link* with initial value equal to {NULL, NULL}. The second attribute defines the subtree-OID. The definitions of the rest of attributes are defined in the “struct subtree” in Chapter 4 and omitted here.

```
struct subtree subtrees[] = {
    {
        {NULL, NULL},
        {MIB, 1}, 7,
        (struct variable *)system_variables,
        sizeof(system_variables)/sizeof(*system_variables),
        sizeof(*system_variables)
    },

    {
        {NULL, NULL},
        {SUBAGENTMIB}, 7,
        (struct variable *)subagent_variables,
        sizeof(subagent_variables)/sizeof(*subagent_variables),
        sizeof(*subagent_variables)
    },

    {
        {NULL, NULL},
        {REGISTRATIONMIB}, 7,
        (struct variable *)register_tree_variables,
        sizeof(register_tree_variables)/sizeof(*register_tree_variables),
        sizeof(*register_tree_variables) },
    .....
    .....
};
```

Figure 5.8. Built-in subtree Table

At start-up, the agent will build a *subtreeList* which links the entire entries of the table in the lexicographical order.

When the XMS-SNMP agent receives a request for a Management MIB object from a management station, it first searches the *subtreeList* to locate the *subtree* by comparing the subtree-OID. Secondly, it searches the variable table, the fourth attribute of the *subtree*,

to locate the *variable*. Finally, it calls the MIB access function of the *variable* to access the variable value.

In summary, this Chapter has analyzed the possible approaches to manage different types of mobile agents from the XMS-SNMP agent's point of view. It has designed and implemented a management-MIB which can be used to directly maintain mobile subagents and dynamically registered MIB groups

CHAPTER 6

DPI API, AFW, AND APPLICATIONS

This chapter is focused on the application side of the XMS-SNMP agent's DPI interface. Section 6.1 provides a set of DPI application program interfaces (APIs) for subagent, either mobile or non-mobile, applications. These DPI APIs hide the details of protocol level implementation. DPI subagent developers can directly call these APIs to communicate with the XMS SNMP agent.

Section 6.2 provides a DPI application framework (AFW) based on the DPI APIs. This AFW partitions a DPI subagent application into three function areas: a DPI Interface, a subagent engine, and one or more MIB groups. By using this framework, subagent developers only need to subclass the MIBGroup class to implement application specific MIB variable access routines. Section 6.3 provides a MIB group sample which demonstrates how to subclass the MIBGroup class.

Section 6.4 provides an application sample which demonstrates how to build a DPI application by using the DPI APIs and AFW.

The DPI APIs, AFW and application examples are entirely implemented in Java.

6.1 DPI Application Program Interface

6.1.1 Overview

The DPI application program interfaces are abstracted by the *DPIPacket* class, *DPIPacket* subclasses, *VarBind* class, and *VarBindSet* classes. The class hierarchy is depicted in Figure 6.1.

The *DPIPacket* class is responsible for the maintenance of DPI header information.

Inherited from the *DPIPacket* class are a few DPI control packet classes and a *DPIDataPacket* class. The DPI Control packet classes include *DPIOpenPacket*, *DPIClosePacket*, *DPIRegisterPacket*, *DPIUnregisterPacket*, and *DPIAreYouTherePacket*. The major difference between these DPI Control Packet classes and the *DPIDataPacket* class is that the former do not have the variable-binding list while the later does.

Figure 6.1 DPI Packet Classes Hierarchy

The *DPIDataPacket* class contains one or more *VarBind*. The *VarBind* class and its subclass *VarBindSet* class represent the variable-binding of the DPI protocol. The responsibility of the *VarBind* class is for the maintenance of the “group OID” and the “instance OID” of a variable of a DPI *get/getnext* request. The extended responsibility of the *VarBindSet* class is for the maintenance of the “type”, “length”, and “value” of a

variable of a *DPI set/commit/action/undo* request.

Inherited from the *DPIDataPacket* class are *DPIRequestPacket* class, *DPIResponsePacket* class, and *DPITrapPacket* class. The *DPIRequestPacket* class abstracts the behavior of *DPI get*, *getnext*, *set*, *commit*, *undo*, and *action* requests.

The DPI packets can be classified into incoming packets from the XMS-SNMP agent and outgoing packets to the XMS-SNMP agent. Packet classes representing the incoming packets include *DPIUnregisterPacket*, *DPIClosePacket*, *DPIRequestPacket*, and *DPIResponsePacket*. Each of these packet classes has a *parse()* method. The *parse()* method extracts the appropriate bytes from the incoming octet stream and converts them into corresponding attributes of the object.

Packet classes representing the outgoing packets include *DPIOpenPacket*, *DPIRegisterPacket*, *DPIUnregisterPacket*, *DPIAreYouTherePacket*, *DPIClosePacket*, *DPIResponsePacket*, and *DPITrapPacket*. Each of these outgoing packet class has a *getBytes()* method which is used to serialize the object when sending the object to the XMS-SNMP agent.

Packet classes representing both incoming and outgoing packets are *DPIResponsePacket*, *DPIUnregisterPacket*, and *DPIClosePacket*. Each of these packet classes has both *parse()* method and *getBytes()* method.

In the following sub-sections, each packet class is described in more detail.

6.1.2 DPIPacket Class

DPIPacket class is an abstract class. Subagent developers do not use this class directly. However, understanding this class will help subagent developers to correctly use the subclasses of the *DPIPacket* class.

Constructor:

DPIPacket (short id, byte type)

This constructor is intended to be used by the outgoing *DPIResponsePacket*

class to set the packet ID and packet type attributes. Parameter ID is usually obtained from the incoming request packet such as *DPIRequestPacket*.

DPIPacket (byte type)

This constructor does not have the packet ID parameter. The packet ID is uniquely assigned internally. The static attribute, *lastId*, in the DPIPacket class will be automatically incremented every time this method is called. The intended users for this method are all its subclasses except the outgoing *DPIResponsePacket* class.

Methods:

public short id()

This method is used to return the packet ID.

public byte type()

This method is used to return the packet type.

protected int getBytes (byte[] buffer)

This method converts the attributes about the DPI header information into a byte stream and places them in the *buffer*. The returned value is the number of bytes in the buffer. This method is intended to be overridden by its subclasses to continuously serialize the rest of the attributes.

protected void parse (byte[] buffer)

This method extracts the header information from the byte stream in the buffer. It is intended to be overridden by its subclasses to continuously parse the rest of the bytes.

6.1.3 DPIOpenPacket Class

This class is responsible for the maintenance of the DPI open packet information. A subagent will use this class when it needs to “open a connection” with the XMS-SNMP agent.

Constructor:

DPIOpenPacket(String oid, String description, int timeout, int maxVarBinds, int character_set, String password)

This constructor is used for a subagent to create a DPI Open Packet with parameters as defined in the DPI protocol. The packetId will be uniquely assigned by its super class.

Methods:

public int getBytes(byte[] buffer)

This method is used to serialize a DPIOpenPacket. The serialized byte stream is placed in the *buffer*, and the returned value is the number of bytes in the buffer.

public short id()

This method is inherited from its super. It is used to return the packet id when a subagent wants to send an open packet to the agent. A subagent needs to hold this packet ID so that it can be compared with the ID of a received response packet later.

public byte type()

This method is also inherited from its super. It is used to return the packet type.

6.1.4 DPIClosePacket Class

This class is bidirectional. A subagent can both send and receive this packet.

Constructors:

DPIClosePacket (byte reason)

This constructor is used to create an instance of the class with parameter *reason*.

DPIClosePacket()

This constructor is used to create an instance of the class with default reason,

SNMP_CLOSE_goingDown.

Methods:

public byte getReason ()

This method returns a close reason code. It is usually called by a subagent when it receives a DPI Close Packet from the agent.

public String getReasonString ()

This method returns the string format of the close reason code. It is usually called by a subagent when it receives a DPI Close Packet from the agent.

public int getBytes(byte[] buffer)

This method serializes the close packet and places the bytes into the *buffer*. It returns the number of bytes in the buffer. This method is used when the subagent sends a close packet to the agent.

public void parse (byte[] buffer)

This method extracts the DPI close packet attributes from the buffer. A subagent will use this method when it receives an octet stream from the agent.

public byte type()

This method is inherited from its super. It is used to return the packet type.

6.1.5 DPIRegisterPacket Class

This class is used by a subagent to register a MIB group.

Constructors:

DPIRegisterPacket(int timeout, int priority, String groupId, byte view_select, byte bulk_select)

This constructor is used for a subagent to create a register packet with parameters as defined in the DPI protocol.

DPIRegisterPacket (int timeout, int priority, String groupId)

The usage of this constructor is the same as the above except it uses default

view_select and *bulk_select* values which are “disabled”.

Methods:

public int getBytes(byte[] buffer)

This method serializes the object into a byte stream and places the bytes into the buffer. It returns the number of bytes. This method is used when a subagent sends a register packet to the agent.

6.1.6 DPIUnregisterPacket Class

This class is bidirectional. A subagent can both send and receive this packet.

Constructors:

DPIUnregisterPacket()

This constructor is used to create an instance of the class without parameters.

DPIUnregisterPacket(byte reason, String groupOid)

This constructor is used to create an instance of the class when a subagent wants to “unregister” a MIB Group, *groupOid*, with a reason code, *reason*.

DPIUnregisterPacket(String groupOid)

This constructor is used to create an instance of the class when a subagent wants to “unregister” a MIB Group, *groupOid*, with default reason code which is “SNMP_UNREGISTER_justUnregister”.

Methods:

public int getBytes(byte[] buffer)

This method serializes the object and places the bytes into the *buffer*. It returns the number of bytes in the *buffer*. This method is used when a subagent sends the unregister packet to the agent.

public void parse(byte[] buffer)

This method extracts the DPI unregister packet attributes from the *buffer*. It is used when it receives an unregister packet in octet stream from the agent.

public byte getReason()

This method is used for a subagent to get an “unregister” reason.

public byte getReasonString()

This method is used for a subagent to get the “unregister” in a descriptive format.

6.1.7 DPIAreYouTherePacket Class

A subagent uses this class to test whether the connection to the agent is broken.

Constructor:**DPIAreYouTherePacket()**

This constructor is used to create an instance of the class.

Methods:**public int getBytes(byte[] buffer)**

This method serializes the object and places the bytes into the buffer. It returns the number of bytes in the *buffer*. This method is used when a subagent sends the are-you-there packet to the agent.

6.1.8 VarBind Class

This class is responsible for keeping track of a MIB variable’s the Group ID and Instance ID.

Constructors:**VarBind()**

This is the default constructor is used to create an instance without parameters.

Attributes of the object will be filled by parsing the byte stream from the agent.

VarBind(String group, String instance)

This constructor is used to create an instance of the class with group ID, *group*, and instance ID, *instance*.

Methods:

public int getBytes(byte[] buffer, int offset)

This method is used to serialize the object and put the bytes into the buffer starting at position *offset*. The returned value is the next available position in the buffer.

public int parse(byte[] buffer, int offset)

This method parses a byte stream starting from the position of *offset*. The returned value is the next parsing position in the buffer.

public String getGroup()

This method returns the group ID of the variable.

public String getInstance()

This method returns the instance ID of the variable.

public String getOid()

This method returns the object ID which consists of the group ID and instance ID of the variable.

6.1.9 VarBindSet Class

This class extends the VarBind class. It has the following added constructors and methods:

Constructor:

VarBindSet()

This constructor is used to create an instance of the class without parameters.

VarBindSet(String group, String instance, byte SNMPTType, short length, byte value[])

This constructor is used to create an instance of the class with full parameters: the variable's group ID, instance ID, value, type, and length.

VarBindSet(VarBindSet vbs)

This constructor creates an instance of the class out of another instance, *vbs*.

Methods:

public int getBytes(byte[] buffer, int offset)

This method is used to serialize the object and put the bytes into the buffer starting from position *offset*. The returned value is the next available position in the buffer.

public int parse (byte[] buffer, int offset)

This method is used to parse and extract the attributes of the object from a byte stream starting from the position of *offset*. The returned value is the next parsing position in the buffer.

public byte getType()

This method is used to return the type of this variable.

public byte[] getValue()

This method is used to return the value of the variable in byte stream format

public void setValue(byte[] value)

This method is used to set the value of this variable to *value* in byte stream format

public short getLength()

This method returns the length of this value in number of bytes.

6.1.10 DPIRequestPacket Class

This class abstracts request packets from the agent. The primary method is *parse()* which extracts and converts a byte stream into its corresponding attributes.

Constructor:

DPIRequestPacket()

To create an instance without parameters. Attributes of the instance will be filled when parsing its byte stream.

Methods:

public void parse (byte[] buffer)

This method is used to extract attributes of the object from the byte buffer.

public String getCommunity()

This method returns the community name of the request.

public int getCommunityLength()

This method returns the community name length.

public VarBind[] get()

This method is inherited from its super. It returns the variable binding of the request packet.

6.1.11 DPIResponsePacket Class

A subagent can both receive and send objects of this class. The way of using the methods of the class depends on the direction of the packet. When a subagent needs to send a response to the agent, it should first call the constructor having parameters to create a DPIResponsePacket object and then call the serialization method *getBytes()* to convert the object into a byte stream. When a subagent receives a response packet from the agent in byte stream format, the subagent should create an object with default constructor first and then call the *parse()* method to convert the bytes into appropriate attributes of the object.

Constructor:

DPIResponsePacket()

This default constructor is used to create an instance of the class without parameters.

DPIResponsePacket(short id, int errorCode, int errorIndex, int n, VarBind[] vb)

This method is used to create an instance of the class when replying a request from the agent. Parameter *id* should be obtained from the incoming

DPIRequestPacket. Parameter n is the size of the vb parameter.

Methods:

public void parse (byte[] buffer)

This method is used to parse an incoming byte stream in the buffer

public int getBytes(byte[] buffer)

This method is used to serialize the packet into a byte stream in the buffer.

public byte errorCode()

This method is used to return the error code of the response packet.

public int errorIndex()

This method is used to return the error index of the response packet.

public VarBind[] get()

This method is inherited from its super. It returns the variable-binding list of the response packet.

6.1.12 DPITrapPacket Class

This class is used for a subagent to send a trap message to the agent. The getBytes() method is used for the subagent to convert the structured object into a an array of bytes.

Constructors:

DPITrapPacket(int genericCode, short specificCode, String enterpriseId, int n, VarBindSet[] vb))

This constructor is used to create an instance of the class with parameters as defined in the DPI protocol. Packet id will be automatically and uniquely assigned by its super class

Methods:

public int getBytes(byte[] buffer)

This method is used for a subagent to serialize the trap packet into an array of bytes in the *buffer*.

6.2 DPI Subagent Application Framework

6.2.1 Overview

The DPI subagent application framework is depicted in Figure 6.2. It consists of a DPI interface, a DPI subagent engine, and one or more MIB groups. The DPI interface is used to convert a DPI packet into a byte stream and send the bytes to the agent. It is also

Figure 6.2 DPI Application Framework

used to receive a DPI packet in byte stream format and convert it into a DPI packet object. The DPI interface class hides the underlying transport layer details.

The subagent engine is the core of the subagent. When it is started, it sends an “open” request to the agent followed by a number “register” requests for registering the MIB groups. Thereafter, the subagent engine enters a forever loop where it receives requests from the agent, accessing MIB variables in the MIB groups, and send responses back to the agent. It also processes any requests from any MIB group.

Each MIB group is application dependent. The MIBGroup class is an abstract class which captures the generic behavior of a managed MIB group. Application dependent MIB group implementations should extend this class.

The following subsections provide more details for each of the components.

6.2.2 DPIInterface Class

Constructor:

DPIInterface (String host, int port) throws SocketException

This constructor is used to create an instance of the class. The two parameters *host* and *port* are used to identify the agent's hostname and receiving port respectively.

Methods:

public void send (DPIPacket p) throws IOException

This method is used to convert any DPIPacket, including its subclass object, into a datagramPacket and send it to the agent.

public DPIPacket receive() throws IOException

This method is used to receive incoming messages in byte stream format and returns a corresponding object of the *DPIPacket* class. The *DPIPacket* class is an abstract class. The *DPIPacket.type()* method can be used by applications to determine the specific type of the packet. The expected packet classes include *DPIRequestPacket*, *DPIClosePacket*, *DPIUnregisterPacket*, and *DPIResponsePacket*.

public void close()

This method is used to free the socket used by the object.

6.2.3 MIB Group Class

The *MIBGroup* class has three variables, including *group ID*, *priority*, and *time-out*, which are from the DPI protocol. It also has two additional attributes: *registerStatus* and *requestStatus*. The *registerStatus* attribute reflects if the MIB group is registered with the master agent. The *requestStatus* reflects if the MIB group needs to send any request to the agent because of application specific operations. A MIB group can send the following requests to the agent: "register", "unregister", "close", "areYouThere", and "trap".

When a DPI subagent engine is started, it will first read each MIB Group's *group ID* and *priority* by calling the *corresponding groupOid()* and *getPriority()* method, and then create a *DPIRegisterPacket* object and send it to the agent. Upon receiving the response for the request, the DPI subagent engine will set the MIB Group's *priority* to the granted priority and the *registerStatus* to "true" by calling the *setPriority()* and *setRegisterStatus()* method respectively.

When the DPI subagent engine receives a *DPIUnregisterPacket* object from the agent, it will find the MIB Group and set the *registerStatus* to "false" by calling *registerStatus()* method.

Constructor:

MibGroup(String group, int priority, int timeout)

This constructor is used to create an instance of the class with *groupOid*, *priority*, and *timeout* as its parameters. When a MIB Group object is created, its default *registerStatus* is "false", which means it is not registered.

Methods:

public void setRegisterStatus()

This method sets the register status of this MIB group to "true". It can be called by applications as well as the DPI subagent engine.

public void resetRegisterStatus()

This method is used to set the register status of this MIB group to "false". It can be called by applications as well as the DPI subagent engine.

public byte getRequestStatus()

This method returns the current request status of the MIB group. It can be one of the following values: {*unregister*, *register*, *close*, *are_your_there*, *trap*, *none*}. A DPI subagent engine can call this method to determine if this MIB group needs to send requests to the agent.

public void resetRequestStatus()

This method set the request status to “*none*” no matter what the value it was before. A DPI subagent engine calls this method when it completes sending a request to the agent on behalf of the MIB group.

public void setRequestStatus(byte status)

This method is used by applications to set the *requestStatus* to a specific value *status*.

public int getPriority()

This method returns the priority of this MIB group.

public void setPriority(int priority)

This method is usually called by a DPI subagent engine to set the priority to the agent granted value after a successful registration for the MIB group.

public String getGroup()

This method returns the group ID of this MIB group.

public VarBindSet get(VarBind vb)

This method is intended to be overridden by its subclasses to return the variable binding for the *vb* parameter.

public VarBindSet getNext(VarBind vb)

This method is intended to be overridden by its subclass to return a variable binding of an object next to the OID given by the *vb* parameter.

public byte set(VarBindSet vb)

This method is used by a DPI subagent engine to set the variable defined by the *vb* parameter.

public byte commit(VarBindSet vb)

This method is used by a DPI subagent engine to commit a set.

public byte undo(VarBindSet vb)

This method is used by a DPI subagent engine to undo a set.

public byte action(VarBindSet vb)

This method is called by a DPI subagent engine. Applications can take this opportunity to perform some actions after the set/commit/action chain commands.

6.2.4 DPI Subagent Engine Class

Constructor:

DPISubagent(String name, String subagentOid, DPIInterface dpiIf, MibGroup[] mibGroups, int timeout)

This constructor is used to create an instance of the class. Parameter *Name* is a user friendly name for this subagent. Parameter *subagentOid* uniquely defines this subagent. Parameter *dpiIf* is used to send and receive DPI packets to and from the agent. Parameter *mibGroups* is an array of MibGroup variables to which the DPI subagent engine will access.

Method:

public void run()

When it starts, it performs the following:

1. create an object of *DPIOpenPacket*, and send it to the agent.
2. create a *DPITrapPacket* object and send it to the agent.
3. create a *DPIRegisterPacket* object for each mibGroup and send it to the agent.
4. enter a forever loop:

receive DPI Packet through *dpiIf.receive()*

process the received DPI packet

check and process requests from the registered MIB Groups

6.3 MibGroup Subclass Sample ~~MibGroupSample~~

This section describes how to write *MibGroup* subclasses by using a simple example: *MibGroupSample*. In general, a *MibGroup* subclass is required to write an application- dependent constructor, and to override the following methods: *get*, *getnext*, *set*, *commit*, *action*, *undo*.

6.3.1. *MibGroupSample* Attributes and Constructor

Figure 6.3 shows the attributes, constructor, and initialize code. In the sample, two private variables are used. The first variable is *vars* which is an array of three elements of *VarBindSet* type. Each element has its instance ID, type, value, and length. The first element is a read-write integer which can take any of the values {0, 1, 2, 3, 4, 5}. These values represents {REQUEST_NONE, REQUEST_REQISTER, REQUEST_UNREGISTER, REQUEST_CLOSE, REQUEST_ARE_YOU_THERE, REQUEST_TRAP} respectively. The second element is a read-only string, and the third element is a read-only integer.

The second variable, *value1_backup*, is used to backup the first element when changing the value.

Constructors are usually application dependent, but these three parameters, including *groupID*, *timeout*, and *priority*, are mandatory and has to be passed to its super. In the sample, the constructor use the above three parameters. After calling its super, the constructor calls the private *init()* method to initialize application specific attributes.

The private *init()* method initialize the three elements in *vars*. Each element is assigned an instance ID, a type, a length, and an initial value. The first element is assigned with instance ID "1.0", type "integer", length 4, and initial value "0". The second element is assigned with instance ID "2.0", type "string", length of the string, and initial value "DPI subagent string testing". The third element is assigned with instance ID "3.0", type integer, length 4, and initial value 3333.

```

//MibGroupSample.java

import MibGroup;

public class MibGroupSample extends MibGroup{

    // vars are place holder for three objects
    private VarBindSet[] vars = new VarBindSet[3];

    // value1_new is a private variable used when setting the first object.
    private int value1_backup = 0;

    // The constructor is hardcoded to create one MIB group with three instance

    MibGroupSample(String groupId, int priority, int timeout) {
        super(groupId, priority, timeout);
        init();
    }

    // application specific initialization

    init(){

        // create the first object:
        vars[0] = new VarBindSet(groupId,
                                "1.0", // instance
                                DPIValueType.DPI_TYPE_Integer32, // type
                                (short)4, // len
                                intToBytes(0)); // value

        // create the second object:
        String value2 = "DPI subagent string testing";
        vars[1] = new VarBindSet(groupId,
                                "2.0", // instance OID
                                DPIValueType.DPI_TYPE_DisplayString, // variable type
                                (short) value2.length(), // variable len
                                value2.getBytes()); // variable value

        // create the third object
        vars[2] = new VarBindSet(groupId,
                                "3.0", // instance OID
                                DPIValueType.DPI_TYPE_Integer32, //type
                                (short) 4, //len
                                intToBytes (3333) ); // value

    };
}

```

Figure 6.3 MibGroupSample Constructor

6.3.2 “get” Method

Figure 6.4 shows the “*get*” method. The parameter *vb* is a type of *VarBind* which contains the group ID and instance ID. The method will first do a quick check and return null if the requested group ID is not equal to this group ID, or the length of the instance ID is not equal to 3. Then the method will search each elements in the *vars* array to find a match with the instance ID. The method will create and return a new copy of the variable if it finds a match, or return null otherwise.

```
// this method implements the "get" to return object defined in vb.
public VarBindSet get(VarBind vb)
{
    short length;
    byte[] value;
    byte type;

    String instance= vb.getInstance();

    if(vb.getGroup().equals(getGroup())== false || instance.length() != 3 )
        return null; //A quick check

    for(int i =0 ; i< vars.length; i++){
        if( vb.getInstance().equals(vars[i].getInstance() ) ){
            return new VarBindSet(vars[i]);
        }
    }
    return null;
}
```

Figure 6.4 MibGroupSample “get” Method

6.3.3 “getnext” Method

Figure 6.5 shows the “*getnext*” method. This method searches the elements in the *vars* array to find the first object whose object ID is greater than that of the parameter, *vp*. The method will create and return the copy of the variable when it does find the match or return *null* otherwise.

It should be emphasized that the method uses object ID which comprises group ID and instance ID.

```

// this method implements the "getnext"
public VarBindSet getNext (VarBind vb) {

    for(int i =0 ; i<vars.length ; i++){
        int result = vb.getOid().compareTo(vars[i].getOid() );
        if(result < 0 ){
            return new VarBindSet(vars[i]);
        }
    }
    return null;
}

```

Figure 6.5 MibGroupSample "getnext" Method

6.3.4 "set" Method

Figure 6.6 shows the "set" method. First, the method will check if the request is for the first variable by comparing the group ID and the instance ID of the parameter *vp* with the corresponding group ID and instance ID of the first variable in *vars*. This is because only the first variable is meant to be read-writable. It will return an `SNMP_ERROR_noSuchName` if they are not equal. Secondly, the method will check if the requested variable type is the same as that of the first variable in *vars*. It will return an `SNMP_ERROR_wrongType`, if they are not equal. Third, the method will check if the requested variable length is equal to the length of the first variable. If they are not equal, `SNMP_ERROR_wrongLength` will be returned. Finally the value range of request variable is checked. If the value is not within the defined range, `SNMP_ERROR_badValue` will be returned. If all these checks are passed, the new variable value is saved in *value1_backup*, and `SNMP_ERROR_noError` is returned. The *value1_backup* value will be committed to the first variable in the "commit" method which is described next.

```

public byte set (VarBindSet vb)
{
    if( vb.getGroup().equals(getGroup() ) &&
        vb.getInstance().equals(vars[0].getInstance() ) ){

        if(vb.getType() != vars[0].getType() ){
            return SNMPErrorCode.SNMP_ERROR_wrongType;
        }
        if(vb.getLength() != vars[0].getLength() ) {
            return SNMPErrorCode.SNMP_ERROR_wrongLength;
        }

        value1_backup = bytesToInt(vb.getValue() ); // save new value

        if(value1_backup < 0 || value1_backup > 5 ){
            return SNMPErrorCode.SNMP_ERROR_badValue;
        }
        setState = 1;      // SET
        return SNMPErrorCode.SNMP_ERROR_noError;

    }else{
        return SNMPErrorCode.SNMP_ERROR_noSuchName;
    }
}

```

Figure 6.6 MibGroupSample "set" Method

6.3.5 "commit" Method

Figure 6.7 shows the "commit" method. In addition to the checks as was done in the `set` method, it needs to check if the value to be committed in the parameter is equal to the value of `value1_backup`. If they are not equal, `SNMP_ERROR_inconsistentValue` will be returned.

After the consistent check, the original value in `vars[0]` is saved in `value1_backup` variable and `vars[0]` is set with the new value in parameter `vb`. The backup value is for restoring to its original value in the case when the agent issues an "undo" request.

```

public byte commit (VarBindSet vb) {

    if( vb.getGroup().equals(getGroup() ) &&
        vb.getInstance().equals(vars[0].getInstance() ) ){

        if(vb.getType() != vars[0].getType() ){
            return SNMPErrorCode.SNMP_ERROR_wrongType;
        }
        if(vb.getLength() != vars[0].getLength() ) {
            return SNMPErrorCode.SNMP_ERROR_wrongLength;
        }

        if( value1_backup != bytesToInt(vb.getValue() ) ){
            return SNMPErrorCode.SNMP_ERROR_inconsistentValue;
        }
        value1_backup = vars[0].getValue(); // keep a backup

        vars[0].setValue(vb.getValue()); // commit the change
        setState = 2; // COMMIT
        return SNMPErrorCode.SNMP_ERROR_noError;
    }else{
        return SNMPErrorCode.SNMP_ERROR_noSuchName;
    }
}

```

Figure 6.7 MibGroupSample "commit" method

6.3.6 “action” Method

Figure 6.8 shows the “**action**” method. In addition to the checks as did in the set method, it needs to check if the value in the parameter is equal to the committed value in variable vars[0]. If they are not equal, `SNMP_ERROR_inconsistentValue` will be returned. If they are equal, this Sample performs an action of setting the `requestStatus` of the *MibGroupSample* to this value. Later the subagent engine will check the *requestStatus* of each *MibGroup* and process accordingly. This is designed for testing agent’s function of handling requests of unregister, close, and trap from a subagent. Real application can do something differently.

```

public byte action (VarBindSet vb)
{
    if( vb.getGroup().equals(getGroup() ) &&
        vb.getInstance().equals(vars[0].getInstance() ) ){

        if(vb.getType() != vars[0].getType() ){
            return SNMPErrorCode.SNMP_ERROR_wrongType;
        }
        if(vb.getLength() != vars[0].getLength() ) {
            return SNMPErrorCode.SNMP_ERROR_wrongLength;
        }

        if( vars[0].getValue() != bytesToInt(vb.getValue() ) ){
            return SNMPErrorCode.SNMP_ERROR_inconsistentValue;
        }
        setRequestStatus( (byte) value1_new); // real action
        setState = 3;                          // ACTION

        return SNMPErrorCode.SNMP_ERROR_noError;
    }else{
        return SNMPErrorCode.SNMP_ERROR_noSuchName;
    }
}

```

Figure 6.8 MibGroupSample "action" Method

6.3.7 “undo” Method

Figure 6.9 shows the “undo” method. When the method is called, first it will check if the group ID and instance ID of the variable, *vb*, are equal to the corresponding group ID and instance ID of *vars[0]* respectively. If they are not equal, the method will return *SNMP_ERROR_noSuchName*.

Secondly, the method will undo what have been done. If the preceding operation was *set*, the “undo” method simply re-initializes variable *value1_backup* to 0. If the preceding operation was *commit*, the method will restore the original value saved in variable *value1_backup*. The preceding operation is determined by the private *setState* variable.

Third, the method will re-initialize the *setState* variable to 0 and return with no

error.

```

public byte undo (VarBindSet vb)
{
    if( vb.getGroup().equals(getGroup() ) &&
        vb.getInstance().equals(vars[0].getInstance() ) ){

        if(setState == 1){          // just complete SET
            value1_backup = 0;      //free resource
        }else if(setState == 2){    // already COMMIT
            vars[0].setValue(value1_backup); // restore original value
        }
        setState = 0;              // re-initialize setState
        return SNMPErrorCodes.SNMP_ERROR_noError;
    }else{
        return SNMPErrorCodes.SNMP_ERROR_noSuchName;
    }
}

```

Figure 6.9 MibGroupSample undo method

6.4. DPI Subagent Sample ~~DPI Subagent Sample~~

This section describes an example of how to write DPI subagent applications based the DPI application framework and *MIBGroupSample.java*. The example code is shown in Figure 6.10. Explanations are given to the lines of code in bold face.

The first three lines of code in bold face are used to create three MIB Groups. The parameters for each creation are **group ID**, **priority**, and **time-out** respectively. These three newly created MIB groups are placed in the MIB group array, **mibGroups**.

The fourth line of code in bold face is used to create a DPIIf, namely **dpiIf**, with the agent's IP address and receiving port number. Through this dpiIf, the subagent can send and receive DPI packet transparently.

```

//DPITest.java
import java.io.* ;
import java.net.*;
import DIPacket;
import DPIIf;

import DPIPortQuery;
import DPISubagent;

public class DPITest{

public static void main(String argv[]) {
    String host;
    DPIIf dpiIf;
    MibGroup[] mibGroups= new MibGroup[3];

    if(argv.length != 4){
        System.out.println("Usage: DPITest host subagentName subagentOid
priority");
        System.exit(1);
    }
    host = argv[0];
    if(!ObjectId.parseOid(argv[2])){
        System.out.println("invalid subagentOid");
        System.exit(1);
    }
    int priority = Integer.parseInt(argv[3]);
    mibGroups[0] = new MibSample("1.3.6.1.4.1.2.2.1.6.", priority, 10);
    mibGroups[1] = new MibSample("1.3.6.1.4.1.2.2.1.7.", priority, 10);
    mibGroups[2] = new MibSample("1.3.6.1.4.1.2.2.1.8.", priority, 10);
        /* group id,    priority,    timeout */
    try {

        int agentPort = DPIPortQuery.getDPIPortUDP(host);
        dpiIf = new DPIIf(host, agentPort);
        DPISubagentEngine dpiSubagentEngine = new
            DPISubagentEngine(argv[1], argv[2], dpiIf, mibGroups, 10);
            // subagent name, oid, IF,    ..    timeout
        dpiSubagentEngine.run();
    }
    catch (SocketException ex) {
        System.err.println(ex.toString());
    }
    catch (IOException ex) {
        System.err.println(ex.toString());
    }
    catch (DPIPortNotFoundException ex) {
        System.err.println(ex.toString());
    }
}
}

```

Figure 6.10 DPITest.java Code List

The fifth line of code in bold face is used to create the `dpiSubagentEngine`. The parameters are subagent name, subagent OID, `dpiIf`, `mibgroups`, and overall time-out respectively.

The last line of code in bold face is used to start the `dpiSubagentEngine` by invoking its `run` *method()*.

CHAPTER 7

RDPI API AND APPLICATIONS

This chapter is focused on the application side of the RDPI protocol. Section 7.1 describes the RDPI application program interfaces (APIs). These RDPI APIs are implemented by a few RDPI Packet classes which hides the protocol level implementation details. Section 7.2 describes the RDPI interface which is used by applications to send and receive RDPI packets while hiding the transport layer details. Finally, Section 7.3 describes a RDPI tool set as RDPI application examples. The tool set can be used to send RDPI requests and receive RDPI responses. All the APIs, RDPI interface, and applications are implemented in Java.

The relationship between the RDPI Packets, RDPI interface, and RDPI tool set are graphically shown in Figure 7.1.

Figure 7.1 RDPI Packets, Interface, and Application

7.1 RDPI APIs

The RDPI APIs are a composite of a *RDPIPacket* class, a *RvarBind* class, and their subclasses. The relationship of the classes is shown in Figure 7.2. The *RDPIPacket* is responsible for maintaining the common header information of the RDPI protocol and a variable-binding list. The variable-binding list is represented by an array of *RvarBind* objects.

Figure 7.2 RDPI Packet Classes Hierarchy

Inherited from the *RDPIPacket* class are *RDPIRequest* class and *RDPIResponse* class. The *RDPIRequest* class is designed for abstracting outgoing packets. It has a community name attribute, and a *getbytes* method to serialize its attributes. The *RDPIResponse* class is designed for abstracting incoming RDPI response packets. It has an error code and an error index attributes, and a *parse* method to parse incoming packet.

Inherited from the *RDPIRequestPacket* class are *RDPIGetPacket*,

RDPIGetnextPacket, and *RDPISetPacket* class. These three sub-classes have simple constructors since there is no need to define the packet type parameter.

The *RvarBind* class is responsible for maintaining object identifier only. The objects of this class are used by *get* and *getnext* requests. Inherited from the *RvarBind* class is the *RvarBindSet* class which has additional attributes, including the object's type, length and value. The *RvarBindSet* is used by *set* request and *response* request.

In the following subsections, each class is described in more detail from a user's point of view.

7.1.1 RvarBind Class

This class is used to keep track of a variable, which is used for an out-going *get*/*getnext* request packet.

Constructor:

RvarBind(String objectId)

This constructor is used to instantiate an object of the class.

Method:

public int parse(byte[] buffer, int offset)

This method is intended for its subclass to override only.

public int getBytes (byte[] buffer, int offset)

This method is used to serialize the object. The serialized octets are placed in parameter *buffer* starting from a position specified by parameter *offset*. The returned value is the next available position in the buffer.

public String getObjectId()

This method returns the object ID of the variable.

public short getLength()

This method returns the length of this object ID in number of bytes.

7.1.2 RvarBindSet Class

This class is used to maintain the *set* type of variable-binding.

Constructor:

RvarBindSet()

This constructor creates an empty instance. Its attributes will be filled after extracting and parsing a byte stream in the case of a *RDPIResponse* packet.

RvarBindSet(String objectId, byte type, short length, byte value[])

This constructor creates an instance of the class with full parameters. It is intended to be used at the time when creating a *RDPISetPacket* object.

Method:

public int parse(byte[] buffer, int offset)

This method parses and extracts the attributes from the byte stream in the *buffer* starting from position *offset*. It is intended to be used when receiving an incoming response message.

public int getBytes (byte[] buffer, int offset)

This method serializes the object into a byte stream and places them in the *buffer* starting from position *offset*. It is intended to be used by the *RDPISetPacket* when sending the object to the agent.

public byte getDPIType()

This method returns the type of this variable.

public short getLength()

This method returns the number of bytes of the variable's value.

public void print()

This method prints the content of this object in a user friendly format.

7.1.3 RDPIPacket Class

This class maintains the RDPI packet header information and a variable-binding list.

Constructor**RDPIPacket()**

This constructor creates an empty instance. The attributes of the object are to be filled at the time when extracting and parsing an bytes stream.

RDPIPacket(byte type, VarBind[] vb)

This constructor is intended to be called by its subclass *RDPIRequestPacket* to initialize the attributes. An unique packet ID is assigned to the packet automatically at the time when it is created.

Method**public short getPacketId()**

This method returns the packet ID of the object. The packet ID is used for identifying the correspondence between a request packet and its response packet.

public void parse (byte[] buffer)

This method extracts and parses the header information from the byte stream in *buffer*. The corresponding attributes will be filled after the parsing.

7.1.4 RDPIRequestPacket Class

This class abstracts the general behavior of RDPI get, getnext, and set requests.

Constructor:**RDPIRequestPacket(byte type, String community, RvarBind[] vb)**

This constructor creates an instance with parameters of packet type, community, and the variable-binding list. The packet ID is assigned automatically by the super class of this class. The packet type can be any one of these three constant: `SNMP_RDPI_GET`, `SNMP_RDPI_GETNEXT`, `SNMP_RDPI_SET`.

Method:

```
public String community()
```

This method returns the community name of the packet.

```
public int getButes(byte[] buffer)
```

This method serializes the object and place the bytes in the *buffer*. It returns the number of bytes of serialized octets.

7.1.5 RDPIGetPacket Class

This class is used to creates a *RDPIGetPacket* object. The purpose of adding this subclass is to increase code readability, which will save time for user to look at the packet type value.

Constructor:

```
RDPIGetPacket(String community, RvarBind[] vb)
```

This constructor creates a *RDPIGetPacket* with a community name and a variable-binding list.

7.1.6 RDPIGetnextPacket Class

This class creates a *RDPIGetnextPacket* object. It saves the time for user to look at the packet type value.

Constructor:

```
RDPIGetnextPacket(String community, RvarBind[] vb)
```

This constructor creates a *RDPIGetnextPacket* with a community name and a variable- binding list.

7.1.7 RDPISetPacket Class

This class creates a *RDPISetPacket* object. It saves the time for user to look at the packet type value.

Constructor:

RDPISetPacket(String community, RvarBind[] vb)

This constructor creates a *RDPISetPacket* with a community name and a variable-binding list.

7.1.8 RDPIResponsePacket Class

This class keeps track of the specific attributes of the RDPI response packet in addition to the DPI header information in its super class. It is used by a RDPI application to receive and parse a RDPI response message.

Constructor:

RDPIResponsePacket()

This constructor creates an instance of the class with no parameters. Its attributes are intended to be filled after the parsing of a byte stream.

Method:

public byte getErrorCode()

This method returns the error code of the RDPI response packet.

public String getErrorString()

The method returns the error message in user friendly format.

public int getErrorIndex()

This method returns the error index of the RDPI response packet. This index identifies which variable in the list has an error.

public void parse (byte[] buffer)

This method extracts the packet's attributes from the byte stream in the *buffer*. It is used when receiving a RDPI response message.

public void print()

This method prints the RDPI response message on the screen.

7.2 RDPIInterface Class

The *RDPIInterface* class is designed for users to send a *RDPIRequestPacket* and receive a *RDPIResponsePacket*. It hides the transport layer details.

Constructor:

RDPIInterface(String host, int port) throws SocketException

This constructor creates a *RDPIInterface* object. The *host* and *port* are used to identify the agent's host name and port number.

Method:

**public void send(RDPIRequestPacket rp) throws IOException,
SocketException**

This method is used by RDPI applications to send a *RDPIRequestPacket* to the agent.

**public RDPIResponsePacket receive() throws IOException, SocketException,
UnknownPacketException**

This method is used to receive a RDPI response message and return a *RDPIResponsePacket* object.

public void close()

This method is used to release the internal resource (socket).

7.3 RDPI Applications

This section provides a RDPI tool set as RDPI application examples. This tool set includes *RDPIGet*, *RDPIGetnext*, and *RDPISet*. The tool set is used for sending RDPI request message to the agent, and receiving responses from the agent.

7.3.1 RDPIGet Tool

Figure 7.3 shows the *RDPIGet* tool code list. The tool is used to receive requests from command line, create a *RDPIGetPacket* and send it to the agent, receive the response, and display it on the screen. Attention should be given to the code in bold face. The first line of code in bold face is to create a *RDPIInterface* with a host name, and RDPI port number as parameters. The next four lines are used to create a variable binding list with parameters from the command line. The sixth line of code in bold face creates a *RDPIGetPacket* with parameters of community name and the variable-binding list. The next three lines of code are used to send the *RDPIGetPacket*, receive and print the response respectively.

7.3.2 RDPIGetnext Tool

This tool is the same as the *RDPIGet* tool except one line difference. For completeness, we list the code in Figure 7.4, where the difference is highlighted.

7.3.3 RDPISet Tool

This tool is used to receive a RDPI set request from a command line, create and send a *RDPISetPacket* to the agent, receive and display the response.

The main structure is similar with the *DRDPIGet* tool. The major difference from the *RDPIGet* tool is that the variable binding list is created by using *RvarBindSet*, not *RvarBind*. A separate private method *getVarBind()* is used to parse the command line and create these objects. Figure 7.5 lists the code with the major differences highlighted.

```

// RDPIGet.java

import java.io.* ;
import java.net.* ;
import RDPIGetPacket;
import RDPIInterface;

public class RDPIGet{

public static void main(String argv[]) {
    byte snmpType;
    byte value[];
    int id;
    RDPIInterface rdpi;
    int RDPIPort = 165;

    if(argv.length < 3){
        System.out.println("Usage: java RDPIGet host community oid1 ... " );
        System.exit(1);
    }

    try{
        rdpi = new RDPIInterface(argv[0], RDPIPort);

        RvarBind vb[] = new RvarBind[argv.length-2];
        for(int i = 0; i < argv.length-2; i++){
            vb[i] = new RvarBind(argv[2+i]);
        }

        RDPIGetPacket p = new RDPIGetPacket(argv[1],vb);

        rdpi.send(p);

        RDPIResponsePacket rp = rdpi.receive();

        rp.print();
    }catch (UnknownHostException ex){
        System.err.println(ex.toString());
    }catch (SocketException ex){
        System.err.println(ex.toString());
    }catch (IOException ex){
        System.err.println(ex.toString());
    }catch (UnknownPacketException ex){
        System.err.println(ex.toString());
    }
}
}

```

Figure 7.3 RDPIGet Tool Code List

```

// RDPIGetnext.java

import java.io.* ;
import java.net.* ;
import RDPIGetPacket;
import RDPIInterface;

public class RDPIGetnext{

public static void main(String argv[]) {
    byte snmpType;
    byte value[];
    int id;
    RDPIInterface mdpi;
    int RDPIPort = 165;

    if(argv.length < 3){
        System.out.println("Usage: java RDPIGetnext host community oid1... ");
        System.exit(1);
    }
    try{
        mdpi = new RDPIInterface(argv[0], RDPIPort);
        RvarBind vb[] = new RvarBind[argv.length-2];
        for(int i = 0; i < argv.length-2; i++){
            vb[i] = new RvarBind(argv[2+i]);
        }
        RDPIGetnextPacket p = new RDPIGetnextPacket(argv[1],vb);
        mdpi.send(p);
        RDPIResponsePacket rp = mdpi.receive();
        rp.print();

    }catch (UnknownHostException ex){
        System.err.println(ex.toString());
    }catch (SocketException ex){
        System.err.println(ex.toString());
    }catch (IOException ex){
        System.err.println(ex.toString());
    }catch (UnknownPacketException ex){
        System.err.println(ex.toString());
    }
}
}

```

Figure 7.4 RDPIGetnext Tool Code List

```

// RDPISet.java

import java.io.* ;
import java.net.* ;
import RDPISetPacket;
import RDPIInterface;

public class RDPISet{

public static void main(String argv[]) {
    byte value[];
    int id;
    RDPIInterface mdpi;
    int RDPIPort = 165;
    int varNum;

    varNum= argv.length -2 ;

    if(argv.length < 5 || (varNum != 3*(varNum/3) ) ){
        System.out.println("Usage: java RDPISet host community
                           oid1 type1 value1 ... oid_n type_n value_n");
        System.out.println("Type: i, int, s, string");
        System.exit(1);
    }
    varNum= varNum/3;

    try{
        mdpi = new RDPIInterface(argv[0], RDPIPort);
        RvarBind[] vb = new RvarBindSet[varNum];

        for(int i= 0; i< varNum; i++){
            vb[i]=getVarBind( argv[2+3*i], argv[3+3*i], argv[4+3*i] );
        }

        RDPISetPacket sp = new RDPISetPacket(argv[1], vb);
        mdpi.send(sp);
        RDPIPacket rp = mdpi.receive();
        rp.print();
    }
    catch (UnknownHostException ex){
        System.err.println(ex.toString());
    }catch (SocketException ex){
        System.err.println(ex.toString());
    }catch (IOException ex){
        System.err.println(ex.toString());
    }catch (UnknownPacketException ex){
        System.err.println(ex.toString());
    }catch (RvarBindInputException ex){
        System.err.println(ex.toString());
    }
}
}

```

Figure 7.5 RDPISet Tool Code List

```

static RvarBind getVarBind(String oidStr,String typeStr, String
valueStr)
    throws RvarBindInputException
{
    byte snmpType = DPIValueType.getType( typeStr);

    switch(snmpType){
        case DPIValueType.DPI_TYPE_Integer32:
            int val = Integer.parseInt(valueStr);
            System.out.println("set value = " + val);
            return new RvarBindSet(oidStr, snmpType, (short)4,
                DPIValueType.integer_to_bytes(val));

        case DPIValueType.DPI_TYPE_String:
            return new RvarBindSet(oidStr, snmpType,
                (short)valueStr.length(),valueStr.getBytes() );
        default:
            throw new RvarBindInputException("only support s string,
i int");
    }
}
}

```

Figure 7.5 RDPISet Tool Code List (continued)

CHAPTER 8

TEST DESIGN

Since the work presented in the preceding chapters will be used in the Perpetuum Mobile Procura Agent Project[12] by many mobile agent developers, it is important to make sure that the implementation is correct and robust. Hence, this whole chapter is devoted to the development of test strategy and test cases. In the following sections, Section 8.1 describes the function areas to test; Section 8.2 describes the overall test strategy; and Section 8.3 provides the test cases.

8.1 Test Areas

Table 8.1: DPI Messages for Testing

subagent initiated request		Agent initiated requests	
messages to agent	messages from agent	messages to subagent	message from subagent
open	response	unregister	
register	response	close	
unregister	response	get/getnext	response
close		set/commit/action/undo	response
are-you-there	response		
trap			

The test covers three function areas. The first area is the DPI interface including both the implementation in the XMS-SNMP agent side and the DPI APIs in the application side. The test can be conducted by first sending DPI messages from application side to the

XMS-SNMP agent side, or from the XMS-SNMP agent side to the application side, and then verifying the responses. Table 8.1 shows both subagent initiated requests/responses and agent initiated requests and responses.

The second test area is the RDPI interface which covers both the implementation in the agent side and the RDPI APIs in the application side. The test can be conducted by sending RDPI requests from the application to the agent. Table 8.2 shows the messages that an application can send to the agent and the response messages from the agent.

Table 8.2: RDPI Messages for Testing

application to agent	from agent to application
get	response
getnext	response
set	response

The third test area is the management MIB in the agent. The test will verify the correctness of MIB objects as is defined in Table 8.3.

Table 8.3: XMS-SNMP Agent Management MIB Variables to Test

Subagent MIB	Register Tree MIB
subagentNumber	subagentIndex (1)
subagentIndex	registerTreeIndex (2)
subagentOid	registerTreeOid (3)
subagentDescription(3)	registerTreePriority(4)
subagentOpenTime(4)	registerTreeTimeOut(5)
subagentTimeOut(5)	registerTreeAdminStatus(6)
subagentIpAddress(6)	
subagentPortNumber (7)	
maxNumVarBind(8)	
numbOfRegTrees(9)	
subagentAdminStatus(10)	

8.2 Test Strategy

8.2.1 Test Configuration

The test configuration is depicted in Figure 8.1. The XMS-SNMP agent and SNMP trap tool are installed in *artie*. The *artie* is an isolated machine which can only be accessed from a host called *pegasus*. The Java RDPIs tool, and Java DPI subagent samples are installed in *Pegasus*. The SNMP tools can be run in either *pegasus* or *artie*.

Figure 8.1 Test Configuration

8.2.2 Subagent Samples

Two subagents have been designed for supporting the test. Their attributes are shown in Figure 8.2 and Figure 8.3 respectively.

subagent descriptive name: "subagent sample 1" subagent OID 1.3.6.1.4.1.2.2.1.5.1 subagent overall timeout: 10 number of MIB groups: 3						
group id	instance id			request priority	granted priority	time out
	1.0	2.0	3.0			
1.3.6.1.4.1.2.2.1.6	1	"subagent 1 group 1"	113	100		11
1.3.6.1.4.1.2.2.1.7	1	"subagent 1 group 2"	123	100		12
1.3.6.1.4.1.2.2.1.8	1	"subagent 1 group 3"	133	100		13

Figure 8.2 Subagent Sample 1 Attributes

subagent descriptive name: "subagent sample 2" subagent OID 1.3.6.1.4.1.2.2.1.5.2 subagent overall timeout: 20 number of MIB groups: 2						
group id	instance id			request priority	granted priority	time out
	1.0	2.0	3.0			
1.3.6.1.4.1.2.2.1.6	1	"subagent 2 group 1"	213	0		21
1.3.6.1.4.1.2.2.1.7	1	"subagnet 2 group 2"	223	0		22

Figure 8.3 Subagent Sample 2 Attributes

Each subagent has a descriptive name, an unique OID, an overall timeout, and a number of MIB groups. Each MIB group has a group ID, a request priority, a timeout, and a number of instances. Each instance has an instance ID and an initial value.

For example, in Figure 8.2, the descriptive name is "subagent sample 1". The subagent OID is 1.3.6.1.4.1.2.2.1.5.1. The overall timeout is 10 seconds. The number of

MIB group is 3. For the first MIB group, the group ID is 1.3.6.1.4.1.2.2.1.6. The request priority is 100. The timeout value is 11. This first MIB group has three instances. The instance IDs are 1.0, 2.0 and 3.0 respectively. The initial values are 1, "subagent 1 group 1", 113 respectively.

The first instance is a read-write variable. It can take one of the following values: {NONE(0), REGISTER(1), UNREGISTER(2), CLOSE(3), TRAP(4)}. At the initialization time, this value is set to 1, meaning that this MIB group wants to register with the agent. After the MIB is registered, the subagent engine will change this value to NONE (0). Later if a user changes this variable to UNREGISTER (2), the subagent will send agent an unregister message, and then change this value to NONE(0). When the agent receives the "unregister" request, it will unregister this MIB group and release allocated objects. If a user changes this value to TRAP(0), the subagent engine will send a "trap" message to the agent. When the agent receives the "trap" message, it will convert the DPI trap into SNMP trap and forward the trap message to the management station. If users sets this value to CLOSE (3), the subagent will send a "close" message to the agent and exit.

The second and the third instances are read only.

8.2.3 Simple Test Scenarios

There are four simple test scenarios from which complicated test cases can be made of. The first simple test scenario is to access the dynamically extended MIB groups of the subagents using the RDPI tool. This scenario passes through both RDPI and DPI interfaces. It covers test function area 1 partially, and area 2 completely. The line, labelled as 1, in Figure 8.5 shows the scenario.

The second simple test scenario is to access MIB objects in the agent's management MIB. This scenario covers test function area 2 and 3 partially. The line labeled as 2 in Figure 8.4 shows the scenario.

Figure 8.4 Access Extended MIB and Management MIB from RDPI tools

Figure 8.5 "unregister" and "close" through Management MIB

The third simple scenario is to unregister a certain MIB group or close a subagent from management station. Users can use the RDPI tool or the SNMP tool to set the MIB objects, *registerTreeAdminStatus*, or the *subagentAdminStatus* to 1 in the static MIB in the XMS-SNMP agent. When the agent finds the changes of this MIB variable, it will create a DPI unregister or a close request and send it to the subagent. The agent also releases the resources occupied by these objects. This scenario is shown in Figure 8.5 .

Figure 8.6 Trigger subagents to send request scenario

The fourth simple scenario is to make the subagent send request messages such as “unregister”, “close”, “are-you-there”, and “trap” to the agent. This can be done by using the RDPI tool or SNMP tool to set the first instance of any MIB group in any subagent to the corresponding value. Upon a successful *set*, the subagent will update the *requestStatus* attribute of the MIB group’s super class. The *subagentEngine* periodically checks the

requestStatus of the MIB group. It will create a DPI request and send it to the agent when it finds a change requiring to do so. This scenario is shown in Figure 8.6.

The SNMP tools can be used as an alternative to the RDPI tools.

8.2.4 Composing Complex Test Scenarios

Complex test scenarios can be made of the simple test scenarios. For example, after completing each “register”, “unregister”, or “close” request either through the simple scenario 3 or scenario 4, scenario 1 and 2 should follow to verify if the request succeeds.

8.3 Test Cases

8.3.1 Single Subagent

TC# 1: start a subagent and pulling its objects

step 1. In pegasus, start SubagentSample1 by typing

```
>java SubagentSample1 artie&
```

this process can be seen by typing `ps -ef | grep java`

step 2. Retrieve the objects of the first MIB group by typing

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.1.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.2.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.3.0
```

The result should be 1, “Subagent 1 group 1”, 113 respectively

step 3. Retrieve the objects of the second MIB group by typing

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.1.0 .1.3.6.1.4.1.2.2.1.7.2.0
```

```
> java RDPIGet artie public 1.3.6.1.4.1.2.2.1.7.3.0
```

the result should be 1, “subagent 1 group 2”, 123 respectively

step 4. Retrieve the three objects of the third MIB group by typing


```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.8.1.0
.1.3.6.1.4.1.2.2.1.8.2.0
.1.3.6.1.4.1.2.2.1.8.3.0
```

the result should be 1, “subagent 1 group 3 “, 133 respectively

step 5. Subagent sends are-you-there message to agent

This can be achieved by setting the first variable of any of these three MIB groups to 4

```
> java RDPISet artie private .1.3.6.1.4.1.2.2.1.6.1.0 i 4
```

After the subagent receives a response, the subagent will print it on the screen

step 6. Subagent sends trap message to agent.

This can be achieved by setting the first variable of any of these three MIB groups to 5.

```
> java RDPISet artie private .1.3.6.1.4.1.2.2.1.6.1.0 i 5
```

The trap messages are received by the trap daemon in artie and saved in the /tmp/snmptaplogs file. Read this file you will find the message.

TC# 2: Verify the management MIB about the subagent

step 1. Check the subagent number by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.1.0
the value should be 1.
```

step 2. Check the subagent oid by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.2.1
the value should be 1.3.6.1.4.1.2.2.1.5.1
```

step 3. check the subagent's descriptive name by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.3.1
the value should be "Subagent Sample 1"
```

step 5. check the number of registered MIB group by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.8.1
```

The value should be 3, since the SubgentSample1 has three MIB groups

Step 6. check the maximum number of registered mib groups by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.9.1
```

the value should be 3

step 7. check the three mib group oids by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.3.1.1
```

```
> java RDPIGet artie pulbic .1.3.6.1.3.100.2.1.1.3.1.2
```

```
> java RDPIGet artie pulbic .1.3.6.1.3.100.2.1.1.3.1.3
```

the result should be 1.3.6.1.4.1.2.2.1.6

1.3.6.1.4.1.2.2.1.7

1.3.6.1.4.1.2.2.1.8

step 8. check the priority of each MIB group by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.4.1.1
```

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.4.1.2
```

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.4.1.3
```

The results should be all 100

step 9. check the time out of each MIB group by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.5.1.1
```

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.5.1.2
```

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.5.1.3
```

The results should be all 10

step 10. check the adminStatus of each MIB group by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.6.1.1
```

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.6.1.2
```

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.6.1.3
```

The results should be all 0

TC# 3: unregister a MIB group by a subagent

step 1. trigger the subagent to send an unregister message to agent by typing

```
> java RDPISet artie private .1.3.6.1.4.1.2.2.1.6.1.0 i 2
```

This will set the first variable of the first MIB group to 2 (meaning UNREGISTER)

step 2. retrieve the variables in the first mib group by typing

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.1.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.2.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.3.0
```

Should not get any value, because this MIB group is unregistered.

step 3. retrieve the variables in the second mib group by typing

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.1.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.2.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.3.0
```

The returned value should be 0, "subagent 1 group 2", 123. The second MIB group did not unregister.

step 4. check the number of registered mib group by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.8.1
```

The returned value should be 2, since one has been unregistered.

step 5. check the group OID from agent's Management MIB by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.3.1.1
```

Should not return any value, since it is unregistered.

TC# 4: unregister a MIB group by the agent

step 1. set the adminStatus of the second registration entry to 1 by typing

```
> java RDPISet artie private .1.3.6.1.3.100.2.1.1.6.1.2 i 1
```

This will cause the agent sending an "unregister " message to the subagent.

step 2. retrieve the adminStatus and the MIB group OID of the second registration entry

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.6.1.2
```

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.3.1.2
```

Should not get the value, since it is unregistered.

step 3. verify the number of registered MIB groups by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.9.1
```

The returned value should be 1 now. (have unregistered two so far).

step 4. retrieve the variables from the subagent

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.1.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.2.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.3.0
```

should not get any value, because they have all gone.

TC# 5: subagent sends "close" message to the agent

step 1 set the first variable of the third MIB group of the subagent to 3

```
> java RDPISet artie private .1.3.6.1.4.1.2.2.1.8.1.0 i 3
```

This will cause the subagent sends a close message to the agent, and exit by itself.

step 2 Verify the subagent process is exited by typing

```
> ps -ef | grep SubagentSample1
```

step 3 verify the subagentNumber from the management MIB by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.1.0
```

the returned value should be 0

step 4 verify that the information about the subagent can not be retrieved anymore

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.3.1
```

This is to get the subagent description, but should not return any value.

8.3.2 Multiple Subagents

TC# 6: Agent management MIB verification

Step 1. Start two subagents by typing

```
> java SubagentSample 1 artie &
```

```
> java SubagentSample 2 artie &
```

Step 2. Verify the subagent number by typing

```
> java RDPIGet artie public .1.3.6.1.3.100.1.1.0
```

should return 2

Step 3. Verify the subagent description for each subagent

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.3.1
```

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.3.2
```

The first should return "Subagent Sample 1", and second should return "Subagent Sample 2".

Step 4. Verify the number of registered mib groups for each subagent

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.9.1
```

```
> java RDPIGet artie public .1.3.6.1.3.100.1.2.1.9.2
```

The first value should be 3, and the second should be 2

Step 5. Verify the first MIB group ID, priority, and timeout for the first subagent

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.3.1.1
```

```
.1.3.6.1.3.100.2.1.1.4.1.1
```

```
.1.3.6.1.3.100.2.1.1.5.1.1
```

The returned value should be 1.3.6.1.4.1.2.2.1.6 100 and 11 respectively

Step 6. Verify the second mib group oid, priority, and timeout for the first subagent

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.3.1.2
                                .1.3.6.1.3.100.2.1.1.4.1.2
                                .1.3.6.1.3.100.2.1.1.5.1.2
```

The returned value should be 1.3.6.1.4.1.2.2.1.7 100 and 12 respectively

Step 7. Verify the third mib group oid, priority, and timeout for the first subagent

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.3.1.3
                                .1.3.6.1.3.100.2.1.1.4.1.3
                                .1.3.6.1.3.100.2.1.1.5.1.3
```

The returned value should be 1.3.6.1.4.1.2.2.1.8; 100; 13 respectively

Step 8. Verify the first mib group oid, priority, and timeout for the second subagent

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.3.2.1
                                .1.3.6.1.3.100.2.1.1.4.2.1
                                .1.3.6.1.3.100.2.1.1.5.2.1
```

The returned value should be 1.3.6.1.4.1.2.2.1.6, 99 and 21 respectively.

Step 9. Verify the second mib group oid, priority, and timeout for the second subagent

```
> java RDPIGet artie public .1.3.6.1.3.100.2.1.1.3.2.2
                                .1.3.6.1.3.100.2.1.1.4.2.2
                                .1.3.6.1.3.100.2.1.1.5.2.2
```

The returned value should be 1.3.6.1.4.1.2.2.1.7, 99, and 22 respectively

TC# 7: Pulling subagent supported objects

Step 1. pulling the objects in the first MIB group

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.1.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.2.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.6.3.0
```

The values should be 1, "subagent2 group 1", and 213 which correspond to the variables in Subagent Sample2 because its higher priority

Step 2. pulling the objects in the second MIB group

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.1.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.2.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.7.3.0
```

The values should be 1, "subagent 2 group 2", and 223 which are from SubagentSample1

Step 3. pulling the objects in the third MIB group

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.8.1.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.8.2.0
```

```
> java RDPIGet artie public .1.3.6.1.4.1.2.2.1.8.3.0
```

The values should be 1, "Subagent 1 group 3", 133 which correspond to the variables in Subagent Sample1. Note it is not from the SubagentSample2.

CHAPTER 9

CONCLUSIONS

9.1 Summary

This thesis has successfully integrated the Mobile Code Environment (MCE) of *Perpetuum Mobile Procura Agent Project*[12] with the legacy SNMP agent. On the legacy SNMP side, the public UCD-SNMP agent has been modified to support two lightweight protocols, namely the DPI and RDPI, and a management MIB. This modified SNMP agent is called eXtensible, Mobile Supported (XMS) SNMP agent. Mobile agents can dynamically register new MIB groups with the XMS-SNMP agent by using the DPI protocol. Mobile agents can access MIB objects in the legacy SNMP agent by using the RDPI protocol. The management MIB is used for network management stations to query and control mobile subagents and registered MIB groups.

On the application side, a set of DPI APIs and a set of RDPI APIs have been designed and implemented using Java for mobile agent developers. Furthermore, a DPI application framework (AFW) has been built based on the DPI APIs. The APIs and AFW hide protocol level detail so that application developers can concentrate on their application specific problems. Detailed examples have been developed to demonstrate how to use the APIs and AFW. A thorough test plan has been developed to test the above implementation. Various tests have been conducted to ensure the correctness and robustness of the system. This test plan can also serve as a user guide to the use of the system.

In the integration protocol area, the RDPI protocol has been defined by modifying a subset of data messages of the DPI protocol. The simplicity of RDPI protocol is demonstrated by analytically comparing it with SNMP protocol. In addition, the protocol

mismatch problem between the CMU/UCD-SNMP and the DPI in some SNMP *set* situations has been uncovered and fully analyzed. The problem has been solved by the addition of an *action* packet type to the DPI protocol.

9.2 Future Work

During the course of this work, two major directions have been identified for future research and development. The first direction is to investigate how to integrate the mobile code environment with future developments in the legacy SNMP systems. In our work, the inter-working between the mobile code environment and the legacy SNMP system is based on DPI and RDPI protocols. These protocols consider that the managed resources as a whole are in a single network element. “There is no possibility that any SNMP reference to any object type defined in the MIB could resolve to multiple instances of that type”[7]. But legacy SNMP systems are evolving. As Bob Natale, Chairman of the IETF’s SNMP Agent Extensibility (Agentx) working group, pointed out: the “need for a standard base from which to respond to the growing demand for more flexible and dynamic management of multi-component systems - as opposed to single devices - continued to rise in importance..... And users want - indeed, expect - to be able to manage this collection as a single system[23].” How the mobile code environment interacts with the evolution of legacy SNMP is worth further study. Attention and effort should be given to the future agent extensibility protocols.

The second direction is to develop practical mobile code applications based on the DPI APIs, AFW, and RDPI APIs. A great effort has been made to simplify the use of DPI and RDPI protocols. The results of this effort need to be tested in real applications.

REFERENCES

- [1] A. Bieszczad, B. Pagurek, T. White, C. Schramm, G. Susilo *Programmable networks and their management using mobile code*, System and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, Canada, 1997
- [2] A. Bieszczad, *Towards plug-and-play networks with mobile code*, SCE Technical Report SCE-97-08, Carleton University, March 1997
- [3] Andrzej Bieszczad, Bernard Pagurek, *Application-Oriented Taxonomy of Mobile Code*, SCE Technical Report SCE-97-13, Carleton University
- [4] Dave Bridgham, *An Alternative Perspective on Agent Extensibility*, The Simple times, Vol. 4, No. 2, April, 1996, pp.14 - 16
- [5] Barry Bruins, Cisco Systems, *Some Experiences with Emerging Management Technologies* The Simple times, Vol. 4, No. 3, July, 1996, pp. 6- 8
- [6] Ray Burns, Mary Quinn, FTP Software, *The CyberAgent Framework* The Simple times, Vol. 4, No. 3, July, 1996, pp. 12 - 15
- [7] J. Case, M. Fedor, M. Schoffstal, J. Davin, *A Simple Network Management Protocol*, RFC 1157, May 1990
- [8] SNMPv2 Working Group, J. Case, K. McCloghrie, M. Rose, S. Waldbusser, *Introduction to Community-based SNMPv2*, RFC 1901, January 1996
- [9] Mike Daniele, eSNMP, *An Extensible SNMP Agent*, The Simple times, Vol. 4, No. 2, April, 1996, pp.9-14
- [10] Mike Daniele, Bert Wijnen, Dale Francisco, *Agent Extensibility (AgentX) Protocol*, Version 1, <http://www.scguild.com/agentX>
- [11] Patrick Mullaney, *Overview of a Web-based Agent*, The Simple times, Vol. 4, No. 3, July, 1996, pp.8 - 12
- [12] B. Pagurek, Andrzej Bieszczad, *Perpetuum Mobile Procura Agent Project*, <http://www.sce.carleton.ca/netmanagement>

- [13] Randy Presuhn, *Overview of the AgentX Problem Space*, The Simple Times, Vol. 4, No. 2, April, 1996
- [14] M. Rose, *SNMP MUX Protocol and MIB*, RFC 1227, May 1991
- [15] William Stallings, *SNMP, SNMPv2, and CMIP - the Practical Guide to Network-Management Standards*, Addison Wesley Publishing Company, Reading, Massachusetts, 1993
- [16] Gatot Susilo, *Infrastructure for Advanced Network Management based on Mobile Code*, SCE Technical Report SCE-97-10, Carleton University
- [17] Chris Wellens, Karl Auerbach, *Towards Useful Management* The Simple times, Vol. 4, No. 3, July, 1996, pp. 1 - 6
- [18] B. Wijnen, G. Carpenter, K. Curran, A. Sehgal, G. Waters, *SNMP Distributed Protocol Interface*, Version 2.0, RFC 1952, March 1994
- [19] Y. Yemini, G. Goldszmidt, and S. Yemini, *Network Management by Delegation.*, International Symposium on Integrated Network Management, Washington, DC April 1991
- [20] M. Rose, K. McCloghrie *Structure and Identification of Management Information for TCP/IP-based Internets* RFC 1155, May 1990
- [21] CMU-SNMP implementation, <ftp://ece.ucdavis.edu:/pub/snmp/misc/cmu-snm2.1.2.tar.Z>
- [22] UCD-SNMP implementation, <ftp://ece.ucdavis.edu:/pub/snmp/ucd-snm.tar.gz>
- [23] Bob Natal, *Introduction to Agent Extensibility* The Simple Times, Vol. 4, No. 2, April, 1996
- [24] Marshall T. Rose, *The Simple Book: An introduction to Networking Management*", Prentice Hall, 1996.

APPENDIX A

RDPI PROTOCOL AND SNMP PROTOCOL COMPARISON

As discussed before, the RDPI and SNMP protocols are both used to query and change MIB variables in a XMS-SNMP agent. The difference is that SNMP protocol is used by a network manager which might be in a remote location and the communication between the network manager and the agent might cross networks. The RDPI protocol is used by a mobile agent which is in the same network element as the agent, and the communication between the mobile agent and the SNMP agent is local. Therefore, RDPI should be a lightweight protocol. This appendix analytically compares the RDPI protocol with SNMP protocol, through which the simplicity of the RDPI protocol is demonstrated. The following subsections will compare the two protocols in further detail.

A.1 SNMP Message Definition

SNMP message is shown in Figure A.1. As can be seen, it has five types of PDU including *GET*, *GETNEXT*, *SET*, *RESPONSE*, and *TRAP*.

Figure A.1 SNMP Protocol

Figure A.2 (a) provides the definition of SNMP protocol in ASN.1 format [7]. The SNMP message is defined as a *sequence of version, community, and data*. The type of *version* and the type of *community* are *INTEGER* and *OCTET STRING* respectively. *INTEGER* and *OCTET STRING* are primitives. The *data*'s type is *ANY*.

Figure A.2 (b) defines a type *PDUs* for *data*, which is a *CHOICE* of {*get-request*, *get-next-request*, *set-request*, *trap*}. The types of the choice elements are *GetRequest*-

PDU, GetNextRequest-PDU, GetResponse-PDU, SetRequest-PDU, Trap-PDU

respectively.

Figure A.2 SNMP Message in ASN.1

Figure A.3 SNMP PDU in ASN.1

Figure A.3 (a) shows that the types *GetRequest-PDU*, *GetNextRequest-PDU*, *GetResponse-PDU*, *SetRequest-PDU* are further defined as different *contexts of implicit PDU*. The *PDU* type is in turn, as is shown in Figure A.3 (b), defined as a *sequence of request-ID, error-status, error-index, variable-bindings*. The types for the first three

variables, including *request-ID*, *error-status*, and *error-index*, are INTEGER, and the type for the last variable, *variable-bindings*, is VarBindList.

Figure A.4 SNMP VarBindList and VarBind in ASN.1

Figure A.5 SNMP ObjectName and ObjectSyntax in ASN.1

Figure A.4 (a) shows that the *VarBindList* is defined as a *sequence of VarBind*. The *VarBind*, as is shown in Figure A.4 (b), is further defined as a *sequence of name and value*, where the types for the *name* and *value* are *ObjectName* and *ObjectSyntax* respectively.

Figure A.5 (a) shows that the *ObjectName*'s type is defined as a primitive: *OBJECT IDENTIFIER*. Figure A.5 (b) shows that the *ObjectSyntax*'s type is defined as a *CHOICE of {simple, application-wide}*, where the *simple* and *application-wide* are of type *SimpleSyntax* and *ApplicationSyntax* respectively.

Figure A.6 SNMP SimpleSyntax and ApplicationSyntax in ASN.1

Figure A.7 SNMP Basic Object Types in ASN.1

Figure A.6 (a) shows that *SimpleSyntax* is defined as a *CHOICE* of {*number*, *string*, *object*, *empty*}. The types of these variables are all primitives. Figure A.6 (b) shows that the *ApplicationSyntax* is defined as a *CHOICE* of {*address*, *counter*, *gauge*, *ticks*, *arbitrary*}. Their types are *NetworkAddress*, *Counter*, *Gauge*, *TimeTicks*, and *Opaque* respectively (rfc

1155[20]).

Figure A.7 (a) shows that the *NetworkAddress* is defined as a *CHOICE* of *internet*, whose type is *IpAddress*. The *IpAddress* is defined as a four byte *APPLICATION implicit OCTET STRING* as is shown in Figure A.7 (b). Figure A.7 (b) also shows that the *Counter*, *Gauge*, and *TimeTick* are defined as *application implicit INTEGER*, and the *Opaque* is defined as *application implicit OCTET STRING*.

A.2 SNMP Message Encoding

The Basic Encoding Rule (BER) [15] is used to serialize SNMP messages in the ASN.1-format. Using the BER, each ASN.1 type is encoded into a three-field structure as is shown in Figure A.8. The first field is a tag field which indicates the ASN.1 type. The second field is a length field which indicates how many of the octets that follow make up the value of the ASN.1 type being encoded. The third field is a value field, which is the ASN.1 value encoding.

Figure A.8 BER encoding structure

The tag field has two formats. One is the one-octet format as is shown in Figure A.9 (a). Another is the multiple-octet format as is shown in Figure A.9 (b). Each tag field has three sub-fields, including *class*, *p/c*, and *tag number*. The “*class*” sub-field represents four classes of tags in ASN.1, including *UNIVERSAL*, *APPLICATION-WIDE*, *CONTEXT-SPECIFIC*, and *PRIVATE-USE* classes. The “*p/c*” sub-field, which is one bit, indicates whether the encoding is primitive or constructed. The “*tag number*” is used to identify

object types within each “class”. If a tag number is less than 31, then the one-octet tag field

Figure A.9 BER Tag Field

format is used. Otherwise, the multiple-octet tag field format is used.

Figure A.10 shows the length field. As is shown, the length field has three forms, including single octet, multiple-octet definite form, and indefinite form. If the number of octets of the encoding is less than 128, the single-octet format is used to encode the length. Otherwise, the multiple-octet format is used.

If the multiple-octet definite form is used, the first octet has the high-order bit (bit 7) set to 1. The remaining seven bits comprise a number indicating how many octets follow in the length field. The length is encoded using unsigned integer representation found by

concatenating the octets that follow the initial length octet.

If the indefinite form encoding is used, the first octet has the high-order bit set to one and the remaining bits set to zero. The last two octets are set to zero, marking the end of the encoding. The rest of the octets are used to encode the length of the value of the encoding.

Figure A.10 BER Length Field

It should be pointed out that SNMP specifically disallows the use of the indefinite length form for messages, since the indefinite-length form places a heavy burden on the receiving SNMP entity[24, pp. 181].

Obviously, the BER encoding rule is extremely powerful. Nothing cannot be encoded. But the variable length of the *type field* and the variable length of the *length field* require extra processing power when parsing the SNMP packet.

A.3 SNMP and RDPI Overhead Comparison

In this section, the overhead refers to the extra octets which are used for encoding/decoding purpose only. For example, in SNMP, the *version number* is an integer. To encode the version number, a tag field and a length field are required. The tag field and the length field are counted as overhead, which is two in this case.

As discussed in the preceding section, the *tag field* and the *length field* have a one-octet form and a multiple-octet form. Obviously, the one-octet form has less overhead than the multiple-octet form has. For comparing the overhead with RDPI, the one-octet forms are chosen for the tag field and the length field for SNMP.

Figure A.11 gives the overhead calculation for a SNMP *set* packet. The *set* packet has only one *INTEGER* variable. Both the *tag* field and the *length* field are counted as one octet, the best case for SNMP. The total overhead is 23 octets.

Table A.1 shows the overhead calculation for a RDPI *set* packet. The first two octets for the “packet length to follow”, and the two octets for the “community name length” should be calculated as overhead, since they are used for encoding/decoding purpose only. As with BER encoding, the “variable type” and the “length of value” should also be counted as overhead. The two NULLs terminating the “object ID” should also be counted as overhead, since they are only used for identifying the end of the “object ID”. In total the overhead for the RDPI message is 9 octets for this case packet.

It can be seen that the SNMP *set* packet overhead is 23/9 times of the RDPI *set* overhead for this simple case. For each additional variable-binding, SNMP packet requires minimum 6 octets overhead, while the RDPI packet requires maximum 5 octets. For other types of PDUs such as *get*, *getnext*, the overhead ratios are even higher.

Table A.1: RDPI SET Packet Overhead Calculation

Offset (octet)	Field	overhead (octet)
0	packet length to follow (MSB to LSB)	2
2	protocol version (=2)	
3	protocol minor version (=2)	
4	protocol release (=0)	
5	packet ID (MSB to LSB)	
7	packet type = SNMP_DPI_SET (=3)	
8	community name length (MSB to LSB)	2
10	community name (if any)	
10+L1	object ID with two NULLs	2
10+L2	Variable Type	1
10+L2+1	Length of Value (2 octets, MSB to LSB)	2
10+L2+3	Value	
10+L3	optionally more variable-bindings (object ID, type, length, value)	
note	L1 = length of community name L2 = L1 + strlen(object ID) + 2 L3 = L2 + 3 + length of value	

A.4 SNMP and RDPI Performance Comparison

This section is focused on the efficiency comparison of processing SNMP messages and RDPI messages. Generally speaking, the BER encoding mechanism is not efficient in terms of processing. It is pointed out that “other encoding mechanisms are often 20 (or more) times faster than using the BER” [24, *pp. 129*]. The RDPI protocol is faster than the SNMP protocol from the performance perspective. This is explained in the following subsections.

A.4.1 Header Processing Comparison

Decoding an RDPI header is faster than decoding a SNMP header. The main reason is that the SNMP protocol uses the BER encoding rule which encodes every field in a generic way so that the offset of each field is not in a fixed position, and the size for each tag and length field needs to be determined at run time. In contrast, the RDPI protocol tries its best to fix the offset and the size of each field before hand. For example, in the RDPI header, fixed are the offsets and sizes of these fields: packet length, protocol version, packet ID, and packet type. Therefore, they can be extracted directly. Beside, processing the “SEQUENCE” in the SNMP header is an extra cost (see Figure A.11).

A.4.2 Variable-Binding List Processing Comparison

Decoding an RDPI variable-binding list is faster than decoding a SNMP variable-binding list. This is because the following extra processing is required: one occurrence of variable-binding list definition, n occurrences of variable-binding definition, n occurrences of object name and value decoding, where n is the number of variable-binding in the variable-binding list.

A.4.2.1 SNMP Required Extra Processing

The SNMP variable-binding list requires extra processing for the decoding of one “SEQUENCE OF VarBind” and n “SEQUENCE {name, value}”, where n is the number

of variable-binding in the list (see Figure A.11 for reference). Figure A.12 shows the code fragment of processing of the “SEQUENCE OF” implemented by CMU-SNMP/UCD-SNMP. It first calls `asn_parse_header(data, &length, &type)` to return the type and length, and then verify if the type is “SEQUENCE OF” by comparing it with `(ASN_SEQUENCE | ASN_CONSTRUCTOR)`.

```
/* the following code is from parse_var_op_list() of agent/snmp_agent.c */

data = asn_parse_header(data, &length, &type);
if (data == NULL){
    ERROR("not enough space for varlist");
    return PARSE_ERROR;
}
if (type != (u_char)(ASN_SEQUENCE | ASN_CONSTRUCTOR)){
    ERROR("wrong type");
    return PARSE_ERROR;
}
```

Figure A.12 Processing “SEQUENCE OF”

```
/* the following code is from snmplib/asn1.c */

u_char *
asn_parse_header(data, datalength, type)
    u_char *data; /* IN - pointer to start of object */
    int *datalength; /* IN/OUT - number of valid bytes left in buffer */
/*
    u_char *type; /* OUT - ASN type of object */
{
    register u_char *bufp = data;
    register header_len;
    u_long asn_length;

    /* this only works on data types < 30, i.e. no extension octets */
    if (IS_EXTENSION_ID(*bufp)){
        ERROR("can't process ID >= 30");
        return NULL;
    }
    *type = *bufp;
    bufp = asn_parse_length(bufp + 1, &asn_length);
    if (bufp == NULL)
        return NULL;
    header_len = bufp - data;
    if (header_len + asn_length > *datalength){
        ERROR("asn length too long");
        return NULL;
    }
    *datalength = (int)asn_length;
    return bufp;
}
```

Figure A.13 `asn_parse_header()` Function Call

The implementation of `asn_parse_header(data, &length, &type)` is shown in Figure A.13. The macro `IS_EXTENSION_ID(*bufp)` is defined in Figure A.14. It is used to

check if the tag is an extended object type that requires two bytes. The implementation does not support the type defined by more than one byte. After the check, the object type is extracted by `*type = *bufp;`

/* the following code is from file snmplib/asn1.h */

```
#define ASN_EXTENSION_ID    (0x1F)
#define IS_EXTENSION_ID(byte)    (((byte) & ASN_EXTENSION_ID) == ASN_EXTENSION_ID)
```

Figure A.14 ASN_EXTENSION_ID Macro Definition

/* the following code is from snmplib/asn1.c */

```
/*
 * asn_parse_length - interprets the length of the current object.
 * On exit, length contains the value of this length field.
 *
 * Returns a pointer to the first byte after this length
 * field (aka: the start of the data field).
 * Returns NULL on any error.
 */
u_char *
asn_parse_length(data, length)
    u_char *data;    /* IN - pointer to start of length field */
    u_long *length;   /* OUT - value of length field */
{
    register u_char lengthbyte = *data;

    if (lengthbyte & ASN_LONG_LEN){
        lengthbyte &= ~ASN_LONG_LEN;    /* turn MSb off */
        if (lengthbyte == 0){
            ERROR("We don't support indefinite lengths");
            return NULL;
        }
        if (lengthbyte > sizeof(long)){
            ERROR("we can't support data lengths that long");
            return NULL;
        }
        memmove(length, data + 1, (int)lengthbyte);
        *length = ntohl(*length);
        *length >>= (8 * ((sizeof(int)) - lengthbyte));
        return data + lengthbyte + 1;
    } else { /* short aslength */
        *length = (long)lengthbyte;
        return data + 1;
    }
}
```

Figure A.15 asn_parse_length() Function Call

The `asn_parse_length()` function is defined in Figure A.15. It supports length

size from one byte to sizeof(long) bytes. It does not support infinite bytes length.

Figure A.16 shows the code fragment of processing “SEQUENCE”. Basically it is the same as that shown in Figure A.12, and calls all the functions as shown from Figure A.13 to A.15.

/* the following code is from parse_var_op() of snmplib/snmp.c */

```
data = asn_parse_header(data, &length, &type);
if (data == NULL){
    ERROR("");
    return NULL;
}
if (var_op_type != (u_char)(ASN_SEQUENCE | ASN_CONSTRUCTOR))
    return NULL;
```

Figure A.16 Processing “SEQUENCE”

A.4.2.2 Object Identifier Processing

In the SNMP protocol, each variable-binding is defined as a SEQUENCE {name, value}. The “name” is actually an object identifier. Decoding the object identifier in SNMP is much more complicated than in RDPI. Figure A.17 lists the code fragment which decodes the name in the SEQUENCE {name, value}. It first calls `asn_parse_objid()` to parse the bytes pointed by “data”, and then checks if the returned type is OBJECT IDENTIFIER by comparing it with `(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_OBJECT_ID)`.

/* the following code is from parse_var_op() of snmplib/snmp.c */

```
data = asn_parse_objid(data, &var_op_len, &var_op_type, var_name, var_name_len);
if (data == NULL){
    ERROR("");
    return NULL;
}
if (var_op_type != (u_char)(ASN_UNIVERSAL | ASN_PRIMITIVE | ASN_OBJECT_ID))
    return NULL;
```

Figure A.17 Processing “OBJECT IDENTIFIER ”

Figure A.18 lists the `asn_parse_objid()` code implemented by CMU-SNMP/UCD-SNMP. The object type is extracted by `*type = *bufp++`, which does not consider

the extended format (more than one byte). The object length is extracted by calling `bufp = asn_parse_length(bufp, &asn_length)`, which takes a fair amount of processing.

/* the following code is from snmplib/asn1.c */

```

u_char *
asn_parse_objid(data, datalength, type, objid, objidlength)
    u_char      *data;          /* IN - pointer to start of object */
    int         *datalength;    /* IN/OUT - number of valid bytes left in buffer */
    u_char      *type;          /* OUT - ASN type of object */
    oid         *objid;         /* IN/OUT - pointer to start of output buffer */
    int         *objidlength;   /* IN/OUT - number of sub-id's in objid */
{
    /*
     * ASN.1 objid ::= 0x06 asnlength subidentifier (subidentifier)*
     * subidentifier ::= (leadingbyte)* lastbyte
     * leadingbyte ::= 1 7bitvalue
     * lastbyte ::= 0 7bitvalue
     */
    register u_char *bufp = data;
    register oid *oidp = objid + 1;
    register u_long subidentifier;
    register long length;
    u_long      asn_length;

    *type = *bufp++;
    bufp = asn_parse_length(bufp, &asn_length);
    if (bufp == NULL)
        return NULL;
    if (asn_length + (bufp - data) > *datalength){
        ERROR("overflow of message");
        return NULL;
    }
    *datalength -= (int)asn_length + (bufp - data);

    /* Handle invalid object identifier encodings of the form 06 00 robustly */
    if (asn_length == 0)
        objid[0] = objid[1] = 0;

    length = asn_length;
    (*objidlength)--; /* account for expansion of first byte */
    while (length > 0 && (*objidlength)-- > 0){
        subidentifier = 0;
        do { /* shift and add in low order 7 bits */
            subidentifier = (subidentifier << 7) + (*(u_char *)bufp & ~ASN_BIT8);
            length--;
        } while (*(u_char *)bufp++ & ASN_BIT8); /* last byte has high bit clear */
        if (subidentifier > (u_long)MAX_SUBID){
            ERROR("subidentifier too long");
            return NULL;
        }
        *oidp++ = (oid)subidentifier;
    }

    /*
     * The first two subidentifiers are encoded into the first component
     * with the value (X * 40) + Y, where:
     * X is the value of the first subidentifier.
     * Y is the value of the second subidentifier.
     */
    subidentifier = (u_long)objid[1];
    if (subidentifier == 0x2B){
        objid[0] = 1;
        objid[1] = 3;
    } else {
        objid[1] = (u_char)(subidentifier % 40);
        objid[0] = (u_char)((subidentifier - objid[1]) / 40);
    }

    *objidlength = (int)(oidp - objid);
    return bufp;
}

```

Figure A.18 `asn_parse_objid()` Function Call

Extracting an object identifier is complicated because of its compact encoding rule. Each object identifier consists of one or more sub-identifiers. Each sub-identifier consists of zero or more leading bytes and a last byte. The most significant bit of each leading byte is set to one, while the most significant bit of the last byte is set to zero. Thus, the sub-identifier is represented by concatenating one or more 7-bit values together and treating the resulting string of bits as an unsigned number, which means that extra processing time is required when extracting/assembling them. Besides, the BER places certain restrictions on any object identifier it encodes: the first element must take the value of 0, 1, or 2; and the second element must take a value less than 40 if the first element is 0 or 1 [24, pp. 184]. The first two elements in the sequence of components form a sub-identifier with the value $X * 40 + Y$, where X is the value of the first element, and Y is the value of the second elements. The bottom part of Figure A.18 is the code fragment to decode the first two bytes.

```

/* the following code is from dpi20api/snmp_mdPI.c */

static char *copy(char *cp, int *len)
{
    char *new_p;

    if (cp) {
        *len = strlen(cp) + 1;           /* get length of string */
        new_p = (char *) malloc(*len);  /* get memory for it */
        if (new_p) {                    /* success, so copy it */
            strcpy(new_p, cp);          /* including '\0' char */
        } else {                        /* else error */
            DPI_DEBUG(1, DPI_DM(        /* if debug_lvl >= 1 */
                DPI_M_OUT_OF_MEMORY)); /* give this message */
        } /* endif */
    } else {
        *len = 0;
        new_p = (char *)0;
    } /* endif */
    return(new_p);                      /* return ptr to caller */
}

```

Figure A.19 copy() Function Call

In contrast, in the RDPI protocol, the object identifier is not encoded. It is simply a string with the dots embedded. It can be simply extracted by copying the string. Figure A.19 lists the copy function implemented by DPI20API.

A.4.2.3 Value Processing

Extracting a value's type and length in the RDPI protocol is faster than that in the SNMP protocol. This is because in the SNMP protocol, each value is encoded in the form of {tag, length, value} by using the BER. The type and length fields are extracted by calling function `asn_parse_header()` as listed in Figure A.13 and function `asn_parse_length()` as listed in A.15. In contrast, in the RDPI protocol, since the sizes of each value's type and length are fixed, they can be extracted directly.

In summary, the SNMP protocol is definitely slower than the RDPI protocol in terms of processing. First of all, the SNMP protocol requires extra $(n+2)$ occurrences of processing "SEQUENCEN" or "SEQUENCE OF", which calls `asn_parse_header()` and `asn_parse_length()`, where n is the number of variable-binding in the packet. Secondly, for each object identifier in a variable-binding, the SNMP uses a much more complicated way of encoding/decoding it than the RDPI protocol does. Thirdly, in the SNMP protocol, for each value in a variable-binding, the value's tag and type are variable size, which can only be determined at run time. In contrast, in the RDPI protocol, the sizes of a value's type and length are fixed. Finally, in the header, the SNMP protocol still uses the BER to encode each attribute in a generic way while the RDPI protocol has each field in a fixed position with a fixed size.

The above analysis has only shown that decoding the SNMP protocol is more complicated than decoding the RDPI protocol. In fact, encoding the SNMP protocol is also more complicated than encoding the RDPI protocol. It is also obvious that the larger the number of variable-binding in the packet, the more prominent the efficiency of using the RDPI protocol over the SNMP protocol.

APPENDIX B

THE SNMP AND DPI MISMATCH PROBLEM FURTHER DISCUSSION

This appendix describes the test scenario which uncovered the protocol mismatch problem between the DPI enabled UCD-SNMP and DPI subagents in the case of SNMP *set*. It further analyzes the UCD-SNMP *set* operations in collaboration with DPI protocol and illustrates the rationale of the addition of the *action* message to the DPI protocol.

B.1 Test Configuration and Result

The test configuration is depicted in Figure B.1. The DPI interface has been built into the UCD-SNMP agent. The DPI enabled SNMP agent is in host *artie* and two DPI subagents, namely subagent A and subagent B, are in host *pegasus*. Subagent A and subagent B, have registered the MIB group 1.3.6.1.4.1.2.2.1.5 with the SNMP agent. The priorities assigned to subagent A and B are 100 and 101 respectively. The test objective is to verify if the SNMP agent can correctly handle a *close* request sent from a DPI subagent.

When each of the subagents is started, it performs the following functions:

- sends an *open* request to the SNMP agent to establish a logical connection;
- sends one or more *register* requests to the SNMP agent to dynamically register one or more MIB groups;
- enters a forever loop waiting, receiving, and processing requests from the agent.

In order to make the subagents be able to send *close* packets to the SNMP agent, a read-write variable called *testStr* is defined in each subagent. The instance ID for the variable is 2.0 and the MIB group ID is 1.3.6.1.4.1.2.2.1.5. Therefore, the full object ID for the variable is 1.3.6.1.4.1.2.2.1.5.2.0. This variable can be set to any string value by using SNMP *set* command from an xterm:

```
>snmpset -v 1 host private 1.3.6.1.4.1.2.2.1.5.1.0 s "any string".
```

Each subagent is designed in such a way that when the variable *testStr* is set to a string value “close”, the subagent will send a DPI *close* packet to the agent and exit automatically.

Figure B.1 DPI Protocol Test Configuration

What happened in the test was that every time the variable *testStr* in subagent A, which has the highest register priority 100, is set to “close” from the management station, both subagents exit. This is wrong. Only subagent A is expected to exit.

B.2 UCD-SNMP *Set* Request Processing Analysis

In order to understand the above problem, the UCD-SNMP *set* process should be analyzed. Usually, the SNMP agent daemon is always listening for messages from management stations at the well-known SNMP port (default 161). When it detects that

there are incoming data available at the port, it calls **snmp_read_packet()** routine to read the data. After reading the data, **snmp_read_packet()** routine will call **snmp_agent_parse()** routine to parse the data, and return a response packet. The **snmp_read_packet()** routine sends the response packet back to the management station.

For a SNMP *set* request, **snmp_agent_parse()** will undergo four phases of processing. At each phase, the routine below will be called:

parse_var_op_list(data, length, out_data, out_length, index, pi, action);

where, the parameters have the following meaning:

- *data* is the incoming data packet;
- *length* is the size of the data packet;
- *out_data* is the output after calling this function;
- *out_length* is the size of the out_data;
- *index* is the error index of the variable in the variable-binding list;
- *pi* has the structure packet information;
- *action* is one of the values: {RESERVE1, RESERVE2, COMMIT, ACTION}.

Each time the routine is called, the **action** parameter is passed a different value in the order of RESERVE1, RESERVE2, COMMIT, ACTION.

For each variable in the variable-binding list, **parse_var_op_list()** routine will call the routine below to obtain the variable's value, write_method, and other arguments:

**statP = getStatPtr(var_name, &var_name_len, &statType, &statLen, &acl,
exact, &write_method, pi, &noSuchObject);**

The **getStatPtr()** routine will first search the *subtreeList* [see Figure 4.5] to find the correct *subtree* by comparing the *var_name* with each *subtree*'s ID. Then the routine will search the *variables* of the *subtree* by comparing the rest of the object ID in the *var_name* parameter with each *variable*'s ID. When the *variable* is located, its **findVar()** routine is called to retrieve the *variable*'s value and the *variable*'s write_method.

The returned value is in the form of an array of “unsigned char”. The address of the returned value is assigned to variable *statP*; The length of the value is assigned to argument *statLen*; And the type of the variable is assigned to argument *statType*.

Having completed the execution of the `getStatPtr()`, the `parse_var_op_list()` will call the *variable*’s **write_method** to write the *variable* according to the value of the *action* parameter.

```
err = (*write_method)(action, var_val, var_val_type, var_val_len, statP,
                     var_name, var_name_len);
```

The `(*write_method)()` points to the *variable*’s specific write routine. What it will do with each one of the *action* values, namely RESERVE1, RESERVE2, COMMIT, and ACTION, is application dependent. In general, RESERVE1 and RESERVE2 are intended for applications to reserve some resources before a variable is changed. COMMIT is used for application to commit the change. ACTION gives an application an opportunity to perform some actions following a variable value change.

B.3 The SNMP Set Analysis Before the Addition of ACTION

B.3.1 MIB Access Routines to Subagents

The overall SNMP *set* request processing described in the preceding section still applies to the DPI enabled SNMP agent. When the `getStatPtr()` function searches the *subtreeList* and finds that the variable belongs to a registered MIB group (where *variable_len* is equal to 0), instead of searching the variable array, it will call `findVarInRegSubtree(tp, name, name_len, type, len, acl, write_method, exact, noSuchObject)`. This routine will perform the following functions:

- Make a DPI *get* packet;
- Send the packet to the subagent that has the highest priority for this MIB group;
- Await for response from the subagent;

- Return the value and the `write_method` of the variable.

The `write_method` returned for a registered MIB object is

`writeVarInRegSubtrees(... action)`. This method performs the following functions:

- create a DPI *set*, or DPI *commit*, or DPI *undo* packet according to the value of parameter *action*;
- send the packet to the subagent;
- await for the response from the subagent;
- return the value and control to its caller.

B.3.2 SNMP Agent- DPI Subagent *set* Request Interactions

This sub-section analyzes the detailed interactions between the DPI enabled SNMP agent and DPI subagents. From a management station, the following SNMP *set* request is entered:

```
snmpset -v 1 artie private .1.3.6.1.4.1.2.2.1.5.2.0 s 'close'
```

Where *artie* is the host which has SNMP agent running.

When the SNMP agent daemon in host *artie* detects that there is a message at the SNMP port (default is 161), it calls the routine: `snmp_read_packet()` to read the data. After read the data, the `snmp_read_packet()` routine will call `snmp_agent_parse()` routine to parse the data and obtain the response packet. The `snmp_read_packet()` will send the response packet back to the management station.

The `snmp_agent_parse()` will go four phases of processing. The following subsections will analyze each phase respectively.

B.3.2.1 Phase 1: RESERVE1

At phase 1, the routine below is called with the *action* parameter taking value RESERVE1:

```
parse_var_op_list(data, length, out_data, out_length, index, pi, action)
```

For each variable in the variable-binding list, the `parse_var_op_list()` routine calls

`statP=getStatPtr(var_name,...., &write_method,....)`

to return the variable value, write method, and other attributes. The `getStatPtr()` routine will search the *subtreeList* to find the MIB group that the variable belongs to. If the MIB group is a dynamically registered MIB group, it will call routine:

**`access = findVarInRegSubtree(tp, name, name_len, type, len, acl, write_method,
exact, noSuchObject)`**

This routine creates a DPI *get* packet, sends to the subagent that has the highest registration priority with the MIB group, and awaits for response. When this call is completed, the variable's value is returned in the form of an array of "unsigned char". The address of the returned value is assigned to variable *access*. The write-method returned is **`writeVarInRegSubtrees(... action)`**. The `parse_var_op_list()` routine will call the **`writeVarInRegSubtrees( ... action)`** routine with the *action* parameter equal to `RESERVE1`.

When the **`writeVarInRegSubtrees()`** routine finds that the *action* parameter is `RESERVE1`, it will create a DPI *set* packet, send the packet to the subagent that has the highest registration priority with the MIB group, and await for a response.

On the subagent side, after having "opened" and "registered" with the SNMP agent, the subagent enters a forever loop waiting, receiving and processing requests from the agent.

For the read-write variable, *testStr*, the subagent defines three variables for it internally. This is for backup and recovery purpose. These variables are defined in Figure B.2. Variables *cur_var_p*, *new_var_p*, *old_var_p* represent the current value, new value, and old value respectively.

```

#define testStr      cur_val_p; /*writable object*/
static char          *new_var_p;
static char          *cur_var_p;
static char          *old_var_p;

/* The initial value are defined as the followings */
old_val_p = NULL
cur_val_p = "anything here ";
new_val_p = NULL;

```

Figure B.2 A read-write variable definition in a subagent

When the subagent receives a DPI *set* packet with the variable value equal to “close”, it changes the new_value to “close”, and then send back a success *response*. The variable value change is shown in Figure B.3.

```

old_val_p = NULL;
cur_val_p = "anything here ";
new_val_p = "close";

```

Figure B.3 read-write variable change after a DPI *set*

When the **writeVarInRegSubtrees()** routine receives the DPI *response* packet, it returns control to its caller **parse_var_op_list()**.

When the **parse_var_op_list()** routine has completed all variables in the variable-binding list, it returns control to its caller **snmp_agent_parse()**.

B.3.2.2 Phase 2: RESERVE2

After the control is returned from the first phase, the **snmp_agent_parse()** routine will start its second phase by calling **parse_var_op_list()** with parameter *action* equal to RESERVE2.

As with in phase 1, for each variable in the variable-binding list, the routine will call

```
statP=getStatPtr(var_name,...., &write_method,....);
```

to return the variable value, write method, and other attributes. The `getStatPtr()` routine will search the *subtreeList* to find the MIB group that the variable belongs to. If the MIB group is a dynamically registered MIB group, it will call routine:

```
access = findVarInRegSubtree(tp, name, name_len, type, len, acl, write_method,  
                               exact, noSuchObject)
```

This `findVarInRegSubtree` routine will create a DPI *get* packet, send to the subagent that has the highest priority with the MIB group, and await for a response. When this call is completed, the variable's value is returned in the form of an array of "unsigned char". The address of the returned value is assigned to variable *access*. The write-method returned is `writeVarInRegSubtrees( ... ,action)`. The `parse_var_op_list()` routine will call the `writeVarInRegSubtrees( ...,action )` routine with the *action* parameter equal to `RESERVE2`.

When the `writeVarInRegSubtrees( ... action)` routine finds that the *action* parameter is `RESERVE2`, it simply returns control to its caller, `parse_var_op_list( )`, since there is no correspondence in DPI protocol for `RESERVE2`.

When the `parse_var_op_list()` routine has completed all variables in the variable-binding list, it returns control to `snmp_agent_parse()`.

B.3.2.3 Phase 3: COMMIT

After the control is returned from the second phase, the `snmp_agent_parse()` routine will start its third phase by calling `parse_var_op_list()` with parameter *action* equal to `COMMIT`.

As with in phase 1 and in phase 2, for each variable in the variable-binding list, the routine calls

```
statP=getStatPtr(var_name,....., &write_method,.....);
```

to return the variable value, write method, and other attributes. The `getStatPtr()` routine will

search the *subtreeList* to find the MIB group that the variable belongs to. If the MIB group is a dynamically registered MIB group, it will call routine:

```
access = findVarInRegSubtree(tp, name, name_len, type, len, acl, write_method,  
                             exact, noSuchObject);
```

This *findVarInRegSubtree* routine will create a DPI *get* packet, send to the subagent that has the highest priority with the MIB group, and await for a response. When this call is completed, the variable's value is returned in the form of an array of "unsigned char". The address of the returned value is assigned to variable *access*. The write-method returned is *writeVarInRegSubtrees(... action)*. The *parse_var_op_list()* routine will call the *writeVarInRegSubtrees(... action)* routine with the *action* parameter equal to COMMIT.

When the *writeVarInRegSubtrees(... action)* routine finds that parameter *action* is equal to COMMIT, it will create a DPI *commit* packet, send the packet to the subagent that has the highest registration priority with the MIB group and await for response.

On the subagent side, when the subagent receives the DPI *commit* packet, it will copy the *cur_val_p* into *old_val_p*, and the *new_value_p* to the *cur_value_p*, and nullify the *new_val_p*. The variable value changes are shown in Figure B.4

```
old_val_p = anything here;  
cur_val_p = "close";  
new_val_p = "NULL";
```

Figure B.4 read-write variable change after a DPI *commit*

Having changed the variable values, the subagent will make a DPI *response* packet and send it to the agent. Thereafter, the subagent will check the *cur_val_p* value. When the subagent finds that the *cur_val_p* has been changed to "close", it will create a DPI *close* packet, and send it to the agent. Since there is no response from the agent for the DPI close request [18], the subagent will automatically exit.

On the SNMP agent side, when **writeVarInRegSubtrees()** receives the response for the *commit* from the subagent, it will return control to its caller **parse_var_op_list()**. When **parse_var_op_list()** completes all variables in the variable-binding list, it returns control to **snmp_agent_parse()** routine.

B.3.2.4 Phase 4: ACTION

After the control is returned from phase 3, the **snmp_agent_parse()** routine will start its fourth phase by calling **parse_var_op_list(..., action)** with parameter *action* equal to ACTION.

As with in phase 1, 2 and 3, for each variable in the variable-binding list, the routine calls **statP=getStatPtr(var_name,..., &write_method, ...);** to return the variable value, write method, and other attributes. The **getStatPtr()** routine will search the *subtreeList* to find the MIB group that the variable belongs to. If the MIB group is a dynamically registered MIB group, it will call routine:

```
access = findVarInRegSubtree(tp, name, name_len, type, len, acl, write_method,  
                             exact, noSuchObject)
```

This **findVarInRegSubtree** routine will create a DPI *get* packet, send it to the subagent that has the highest priority with the MIB group, and await for a response.

Since the remote subagent that used to have the highest registration priority with the MIB group does not exist any more, the set time-out interval will expire. Therefore, the **findVarInRegSubtree()** call will fail. Recursively, the caller **parse_var_op_list()** will fail.

When **snmp_agent_parse()** finds that the **parse_var_op_list()** call fails, it will create a failure *response* and return the control to its caller **snmp_read_packet()**. The **snmp_read_packet()** will send it to the NMS.

Having completed the processing of this packet, the SNMP daemon continues to check next available port, which is the DPI port. The DPI port should have the *close* request

packet available. After read the data from the DPI port, the `DPI_read_packet()` will call `snmp_dpi_agent_parse()` function to parse and process. For the DPI *close* request, the agent will remove the *subagent*, the *registerTrees* of the *subagent*, and the dynamically registered MIB groups that has no *registerTree* anymore. In the case of our test configuration, the *registreTree* of subagent A has been removed from the SNMP agent. Therefore, the *registreTree* of subagent B has the highest priority at the point.

Having completed the processing of this DPI packet, the SNMP daemon continues to check next available port, which is the SNMP port. Since on the NMS side, the “UCD snmpset process” is designed to re-send (up to 5 times) the *set* request upon a failure response, the agent receives the same SNMP *set* request.

As with the first SNMP *set* request, the agent repeats the above four phases which makes the next subagent, subagent B in the test configuration, closed.

B.4 The SNMP Set Analysis After the Addition of ACTION

An addition of DPI *action* message to the DPI protocol will solve the above problem. In phase 3, when the subagent receives a DPI *commit* packet from the agent, it should not send the DPI *close* request even if the `cur_val_p` has been changed to “close”. The time for the subagent to send a DPI close packet is when it receives a DPI *action* packet.

This way, in phase 4, the `findVarInRegSubtree()` call will be able to receive a response for the DPI *get* packet from the subagent. Therefore, its caller `getStatPtr()` will succeed. The returned write method will be called with parameter *action* equal to ACTION.

When the `writeVarInRegSubtrees( ... action)` routine finds that parameter *action* is equal to ACTION, it will create a DPI *action* packet, send the packet to the subagent that has the highest registration priority with the MIB group and await for response.

When the subagent receives the DPI *action* request, it will first send a response to

agent. Then it will check the value of `cur_val_p`. If `cur_val_p` equals to “close”, the subagent will create a DPI *close* packet and send it to agent. Thereafter, the subagent exits automatically.

When the **write_method** receives the DPI *response* for the DPI *action* from the subagent, it will return to its caller **parse_var_op_list()** with no error. When every variable has been set without any error, the **parse_var_op_list()** will return control to the caller **snmp_agent_parse()** with success. The **snmp_agent_parse()** routine will create an success *response* packet and return control to its caller **snmp_agent_read()**. The **snmp_agent_read()** function will send the *response* packet to the management station.

On the management station side, when the “snmpset process” receives the response, it will print the result and exit normally.

On the agent side, having completed the processing of this SNMP *set* request, the SNMP daemon continues to check next available port. At the DPI port, there should be a DPI *close* packet sent from the subagent which has already exited. The agent will call **DPI_read_packet()** routine to read the message. After read the data from the DPI port, the **DPI_read_packet** will call *snmp_dpi_agent_parse()* function is to parse and process the data. For the DPI *close* request, the agent will remove the *subagent*, the *registerTrees* of the *subagent*, and the dynamically registered MIB groups that has no *registerTree* anymore.

At this point, the *registereTree* of subagent A has been removed from the SNMP agent. And the subagent B remains.

APPENDIX C

MOBILE AGENT MANAGEMENT-MIB

-- XMS-SNMP Agent Management MIB definition

XMS-SNMP-AGENT-MIB DEFINITIONS ::= BEGIN

IMPORTS

OBJECT-TYPE

FROM RFC-1212

IpAddress, TimeTicks

FROM RFC1155-SMI

DisplayString

FROM RFC1213-MIB

xmsAgentMIB OBJECT IDENTIFIER ::= { 1 3 6 1 3 100 }

subagentMIB OBJECT IDENTIFIER ::= { xmsAgentMIB 1 }

regTreeMIB OBJECT IDENTIFIER ::= { xmsAgentMIB 2 }

--

-- subagent MIB groups

--

-- This MIB group defines the number of the subagents in the agent.

-- and list of subagent entry.

--

subagentNumber OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The number of subagents currently having connection
with the agent"

--DEFAULT 0

::= { subagentMIB 1 }

subagentTable OBJECT-TYPE

SYNTAX SEQUENCE OF SubagentEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"A list of subagent entries. The number of entries
is given by the value of subagentNumber"

::= { subagentMIB 2 }

subagentEntry OBJECT-TYPE

SYNTAX SubagentEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"A subagentEntry containing objects related to the
attributes of subagent."

INDEX { subagentIndex }

::= { subagentTable 1 }

```

SubagentEntry ::= SEQUENCE {
    subagentIndex
        INTEGER,
    subagentOid
        OBJECT IDENTIFIER,
    subagentDescription
        DisplayString,
    subagentOpenTime
        TimeTicks,
    subagentTimeOut
        INTEGER,
    subagentIpAddress
        IpAddress,
    subagentPortNumber
        INTEGER,
    numberOfRegTrees
        INTETER,
    maxNumbOfRegTress
        INTEGER,
    subagentAdminSatus
        INTEGER,
}

```

```

subagentIndex OBJECT-TYPE
    SYNTAX INTEGER
    ACCESS read-only

```

STATUS mandatory

DESCRIPTION

"A unique value for each subagent. Its value ranges
between 1 and the value of subagentNumber"

::= { subagentEntry 1 }

subagentOid OBJECT-TYPE

SYNTAX OBJECT IDENTIFIER,

ACCESS read-only

STATUS mandatory

DESCRIPTION

" sub-agent ID, which is a unique ASN.1 OBJECT identifier"

::= { subagentEntry 2 }

subagentDescripton OBJECT-TYPE

SYNTAX DisplayString

ACCESS read-only

STATUS mandatory

DESCRIPTION

" A desriptive name of the subagent."

::= { subagentEntry 3 }

subagentOpenTime OBJECT-TYPE

SYNTAX TimeTicks

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The time the subagent opens the connection. It is the value of the sysUptime at that point."

::= { subagentEntry 4 }

subagentTimeOut OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

" This is an overall default requested timeout value in seconds to be used for this agents if the register-tree does not define its own timeout value".

::= { subagentEntry 5 }

subagentIpAddress OBJECT-TYPE

SYNTAX IpAddress

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The subagent's IP address is using UDP as transport"

::= { subagentEntry 6 }

subagentPortNumber OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The port number used when the subagent opened the connections. The IP address and the port number uniquely identify the subagent process."

::= { subagentEntry 7 }

numberOfRegTrees OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

"the number of the sub-trees/mib groups currently being registered"

::= { subagentEntry 8 }

maxNumOfRegTree OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The maximum number of mib groups/or registered sub-trees this subagent to register or had ever been registered.

::= { subagentEntry 9 }

subagentAdminStatus OBJECT-TYPE

SYNTAX INTEGER {

open(0),

close(1)

}

ACCESS read-write

STATUS mandatory

DESCRIPTION

"The initial status is 0 which means open. This subagent

can be closed when setting this variable to 1

::= { subagentEntry 10}

--

-- registerTreeMIB group

-- The agent has two types of sub-trees: static sub-tree and dynamic sub-tree.

-- The static sub-tree is built in the agent before the agent is started.

-- The dynamic sub-tree is added to the agent at run-time when the agent

-- receives a "register" request.

--

-- Each dynamic sub-tree can be registered by more than one subagents.

-- Each subagent can register multiple different dynamic sub-trees.

--

-- The register-tree represents a registration entry of a dynamic sub-tree

-- for a subagent.

--

-- This MIB group is a list of register-tree entry. Each entry is uniquely

-- identified by the subagent index and the register-index of the subagent.

--

registerTreeTable OBJECT-TYPE

SYNTAX SEQUENCE OF RegisterTreeEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"A list of registerTree entries. The number of entries is given by the value of numberOfRegisterTrees in the subagentTable."

::= { registerTreeMIB 1 }

registerTreeEntry OBJECT-TYPE

SYNTAX RegisterTreeEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"A registerTreeEntry containing objects related attributes of the registerTree. Each entry is uniquely identified by the subagentIndex and the registerIndex."

INDEX { subagentIndex, registerIndex }

::= { registerTreeTable 1 }

RegisterTreeEntry ::= SEQUENCE {

subagentIndex

INTEGER,

```

registerTreeIndex
    INTEGER,
registerTreeOid
    OBJECT IDENTIFIER,
registerTreePriority
    INTEGER,
registerTreeTimeOut
    INTEGER,
registerTreeAdminSatus
    INTEGER
}

```

subagentIndex OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

"A unique value for each subagent. Its value ranges
between 1 and the value of subagentNumber"

::= { registerTreeEntry 1 }

registerTreeIndex OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

"A unique value for each registerTree in a subagent. "

::= { registerTreeEngtry 2 }

registerTreeOid OBJECT-TYPE

SYNTAX OBJECT IDENTIFIER,

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The oid is the same as the dynamic sub-tree oid which is
registered"

::= { registerTreeEntry 3 }

registerTreePriority OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The priority assigned to this registerTree. If a dynamic
subtree has multiple registerTrees which are registered
by different subagents, only the objects instrumented
by the subagent with highest prioirty can be accessed."

::= { registerTreeEngtry 4 }

registerTreeTimeOut OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

" A timeout value in second that the subagent wishes the agent could wait before declaring time-out when the agent accessing instances instrumented by this subagent."

::= { registerTreeEngtry 5 }

registerTreeAdminSatus OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-write {
 register(0),
 unregister(1)

STATUS mandatory

DESCRIPTION

" The initial vaue is 0 meaing reistered. Setting this variable to 1 will unregister this entry."

::= { registerTreeEngtry 6 }

END

APPENDIX D

AGENT CONFIGURATION

This appendix serves as a user guide to the use of the XMS-SNMP agent. It covers agent configuration, agent startup, and basic agent verification.

D.1 Agent Configuration

Figure D.1 shows the structure of the agent configuration files. The agent daemon is “snmpd”. Environment variables required for running snmpd are stored in “cshrc” file. Related files are sysgroup.ini, v1trap.dst, and mib.txt.

Figure D.1 Agent Configuration File Structure

Figure D.2 shows the content of the `cshrc` file. This file defines a few environment variables: `MIBFILE`, `V1TRAPDEST`, `SYSGROUP`, `V2CONFPATH`.

```
setenv MIBFILE /xuser/public/agent/r3/mib/mib.txt
setenv V1TRAPDEST /xuser/public/agent/r3/etc/v1trap.dst
setenv SYSGROUP /xuser/public/agent/r3/etc/sysgroup
setenv V2CONFPATH /xuser/public/agent/r3/etc
```

Figure D.2 `cshrc` file

Figure D.3 shows the content of the `sysgroup.ini` file. They are the four configuration variables in the system group.

```
sysDescr = artie-for-testing
sysContact = ywang@
sysName = SUN
sysLocation = network managementlab
```

Figure D.3 `sysgroup.ini`

```
134.117.63.6 162 public
```

Figure D.4 `v1trap.dst`

Figure D.4 shows an example of the `v1trap.dst` file. Each line is made up of the IP address, port, and a community name. Agent will sent SNMP version 1 trap to the destination defined by the IP address and the port number. The community name part of the trap message will be filled with community name *"public"*.

D.2 Start Agent

Step 1: change current direction to /xuser/public/agent/r3

```
> cd /xuser/public/agent/r3
```

Step 2: source the cshrc file by typing

```
> source cshrc
```

Step 3: switch to super user,

```
> su root (with root password)
```

Step 4: start the daemon, and exit from the root.

```
> ./snmpd
```

```
> exit (exit root)
```

Step 5: To verify if the snmp daemon is running, type -

```
> ps -ef | grep snmpd
```

You should be able to see the snmpd process.

D.3 Agent Configuration Verification

Step 1: to verify the *sysDescr*, *sysContact*, *sysName*, and *sysLocation*, type

```
>java RDPIGet artie public .1.3.6.1.2.1.1.1.0
```

```
>java RDPIGet artie public .1.3.6.1.2.1.1.4.0
```

```
>java RDPIGet artie public .1.3.6.1.2.1.1.5.0
```

```
>java RDPIGet artie public .1.3.6.1.2.1.1.6.0
```

Step 2: To change *sysDescr*, and verify it, type

```
>java RDPISet artie private .1.3.6.1.2.1.1.1.0 s "new description"
```

```
>java RDPIGet artie public .1.3.6.1.2.1.1.1.0
```

Step 3: To change *sysContact*, and verify it, type

```
>java RDPISet artie private .1.3.6.1.2.1.1.4.0 s "new contact"
```

```
>java RDPIGet artie public .1.3.6.1.2.1.1.4.0
```

APPENDIX E

DPI AND SNMP OBJECT TYPE CONVERSIONS

Table E.1: DPI and SNMP Object Type Conversion

DPI		CMU-SNMP		
value type	value	type	in asn format	value
SNMP_TYPE_Integer32	129	ASN_INTEGER	ASN_INTEGER	0x02
SNMP_TYPE_OCTET_STRING	2	STRING	ASN_OCTET_STR	0x04
SNMP_TYPE_OBJECT_IDENTIFIR	3	OBJID	ASN_OBJECT_ID	0x06
SNMP_TYPE_NULL	4	NULLOBJ	ASN_NULL	0x05
SNMP_TYPE_IpAddress	5	IPADDRESS	(ASN_APPLICATION 0)	0x40 0
SNMP_TYPE_Counter32	134	COUNTER	(ASN_APPLICATION 1)	0x40 1
SNMP_TYPE_Gauge32	135	GAUGE	(ASN_APPLICATION 2)	0x40 2
SNMP_TYPE_TimeTicks	136	TIMETICKS	(ASN_APPLICATION 3)	0x40 3
SNMP_TYPE_DisplayString	9	ASN_OCTET_STRING		(0x04)
SNMP_TYPE_BIT_STRING	10	BITSTRING	ASN_BIT_STR	(0x03)
SNMP_TYPE_NsapAddress	11	NSAP	(ASN_APPLICATION 5)	0x40 5
SNMP_TYPE_UIInteger32	140	UIINTEGER	(ASN_APPLICATION 7)	0x40 7
SNMP_TYPE_Counter64	13	COUNTER64	(ASN_APPLICATION 6)	0x40 6
SNMP_TYPE_Opaque	14	OPAQUE	(ASN_APPLICATION 4)	0x40 4
SNMP_TYPE_noSuchObject	15			1
SNMP_TYPE_noSuchInstance	16			0
SNMP_TYPE_endOfMibView	17			-1

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved