Management of Ubiquitous Networks

by

Catalina Diaz Puig, 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 Applied Science

in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University,

Ottawa, Ontario, Canada, K1S 5B6

May 5th, 2003

© 2003, Catalina Diaz Puig

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

ISBN: 0-612-83472-7

Our file Notre référence

ISBN: 0-612-83472-7

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.

Canada

**The undersigned hereby recommends to
The Faculty of Graduate Studies and Research
acceptance of the thesis,**

Management of Ubiquitous Networks

submitted by

Catalina Diaz Puig,

In partial fulfillment of the requirements
for the degree of Master of Applied Science

A handwritten signature in cursive script, reading "Bernard Pagurek", written over a horizontal line.

Professor Bernard Pagurek, Thesis Supervisor

A handwritten signature in cursive script, written over a horizontal line.

Chairman, Department of Systems and Computer Engineering

Carleton University

May 5th, 2003

Abstract

Ubiquitous networks (UN) are an emerging trend toward numerous, easily accessible computers and computing devices embedded everywhere, in walls, tabletops, and in everyday appliances. Many devices in a UN are resource-constrained, capable-of-communication and mobile. PDAs are very good representatives of resource-constrained network elements of UNs.

Because some of the devices are mobile and the network evolves with time, the network management system (NMS) should provide a mechanism for the dynamic incorporation into and the withdrawal of devices from the network. Hence, it should allow the dynamic extension of the network management information that this demands. The software to be installed in the managed devices must be compatible with the resource-constrained feature of these devices.

The SNMP protocol itself does not provide any mechanism for the dynamic incorporation of network elements and their management information into the network. However, the AgentX protocol (part of the SNMP family) does.

The focus of this thesis is on the development of the necessary software that allows a) the management of the resource-constrained devices in a UN, and b) the management of a UN from a resource-constrained device, like a PDA. For the management of resource-constrained devices I have developed an AgentX subagent: the pdaMIB application. This application is to be installed on the device to be managed. This software allows the dynamic extension of the network management information if a device is added. I have also developed a protocol and two tools that allow the management of a network from a

PDA. For the communication between these tools and the SNMP agent, I have defined and implemented a resource-constrained compatible protocol: the RAgentX (Reverse AgentX) protocol. This protocol is based on AgentX. The RAgentX tools allow a PDA to serve in the capacity of a limited SNMP manager.

In this thesis, I present some tests that have been executed to measure the compatibility of the applications developed with resource-constrained devices. These tests measure response time, bandwidth use, disk space occupation and footprint of the applications. The pdamIB application has been compared with the SNMP agents and AgentX subagents provided by NET-SNMP. The RAgentX tools have been compared to the NET-SNMP tools. These tests show that the applications developed have an 11-20 times smaller footprint and 10-16 times smaller disk space occupation than the other options mentioned above. These tests also show that the processing time and bandwidth use of the applications developed are in the same order of magnitude as the SNMP tools.

If the framework developed in this thesis were to be applied to a UN formed in a household, one would have one ARAX-SNMP agent (AgentX and RAgentX enabled SNMP agent) as the core of the network, and AgentX subagents running on all the managed resource-constrained devices. In addition, there could be one or more RAgentX “managers”. These managers are PDA users that manage the network from a PDA. Overall, this thesis provides a feasible solution for the management of UNs.

Acknowledgements

I would like to thank my thesis supervisor Professor Bernard Pagurek for his invaluable guidance, support and encouragement during these two years.

I am grateful to all my colleagues who are in the Network Management lab for sharing their knowledge with me. I am especially grateful to Vladimir Tosic (a Ph.D. candidate in the Network Management Lab, Carleton University) who provided me a lot of feedback.

Special thanks to my wonderful friends who have always been there when I needed. To those who helped me with my thesis (Anne, Kristen, Tara, Louise, etc.), and those that encouraged me when things got tough...many thanks! You have really made me feel at home - while being away from home!

Me gustaría dar especialmente las gracias a mi familia, que en la distancia ha seguido de cerca mis estudios y mi tesis. Esta tesis va dedicada especialmente a mi padre, que fué el que me impulsó a estudiar ingeniería, y a mi madre, que me apoyó desde el principio en mi "locura" de venirme a Canadá. ¡Muchas gracias!

List of Acronyms

ACM	Access Control Model
AgentX	Agent eXtensibility
ARAX-SNMP	NET-SNMP agent AgentX and RAgentX enable
BER	Basic Encoding Rules
BT	Bluetooth
CLDC	Connected, Limited Device Configuration
CMIP	Common Management Information Protocol
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
GUI	Graphical User Interface
IR	Infrared
IrDA	Infrared Data Association
J2ME	Java 2 Micro Edition
JAX	Java AgentX
JVM	Java Virtual Machine
JXE	J9 eXecutable
KVM	Kilobyte Virtual Machine
LAN	Local Area Network
MIB	Management Information Base
MIDP	Mobile Information Device Profile
NE	Network Element
NET-SNMP	Formerly UCD (Carnegie Mellon University) SNMP agent
NM	Network Management

NMS	Network Management System
OEM	Original Equipment Manufacturer
OID	Object Identifier
OS	Operating System
PAN	Personal Area Network
PDA	Personal Digital Assistant
PDU	Protocol Data Unit
RAgentX	Reverse AgentX
ROM	Read-only Memory
SMI	Structure of Management Information
SNMP	Simple Network Management Protocol
SNMP v1	Simple Network Management Protocol version 1
SNMP v3	Simple Network Management Protocol version 3
TCP	Transmission Control Protocol
TK	Tool Kit
TM1	Transaction Language 1
TMN	Telecommunications Management Network
UC	Ubiquitous Computing
UDP	User Datagram Protocol
UN	Ubiquitous Network
VAME	VisualAge Micro Edition
VarBind	Variable Binding
WEP	Wireless Encryption Protocol

WLAN

Wireless LAN

WRAP

Web Ready Appliance Protocol

Table of Contents

Abstract.....	iii
Acknowledgements	v
List of Acronyms	vi
Table of Contents	ix
List of Figures.....	xii
List of Tables	xv
1. Introduction.....	1
1.1 Background	1
1.2 Motivation.....	8
1.3 Thesis goals.....	11
1.4 Thesis Contributions.....	13
1.5 Thesis Outline.....	17
2. Background Information	18
2.1 Ubiquitous Computing Systems	18
2.1.1 Definition of Ubiquitous Computing.....	18
2.1.2 Challenges in UC Systems.....	20
2.1.3 Research Projects on UC Systems	21
2.2 Issues for Personal Digital Assistants (PDAs) and Other Small Devices.....	22
2.2.1 Memory and Storage Capacity.....	22
2.2.2 Operating systems.....	23
2.2.3 Communication.....	24
2.3 Programming Small Devices.....	28
2.3.1 Java for Small Devices.....	29
2.3.2 J2ME Development Toolkits	33
3. Background Information on SNMP and AgentX.....	36
3.1 The SNMP Protocol	36
3.2 The AgentX Protocol	39
3.2.1 Introduction.....	39
3.2.2 AgentX Protocol Data Units (PDUs)	45
3.2.3 Messages Flow	57

3.2.4 Security	60
3.3 Implementations of AgentX	61
3.4 Java AgentX (JAX)	63
4. The pdaMIB Application	66
4.1 PdaMIB: Application Analysis and Design	67
4.1.1 Application Architecture	67
4.1.2 Main Classes	70
4.1.3 Multi-threading	74
4.1.4 Index Allocation and PDA's Unique Identifier	77
4.2 PDA's MIB	83
4.3 Execution Scenarios	85
4.4 Modifying JAX for Small Devices	93
4.5 Development and Implementation Environment	96
4.6 Limitations	98
4.7 Summary	99
5. The RAgentX protocol and the RAgentX tools	101
5.1 RAgentX Protocol	101
5.1.1 RAgentX PDUs	102
5.1.2 Message Flow	103
5.1.3 Processing RAgentX messages	105
5.2 Integrating RAgentX in the NET-SNMP Agent: the ARAX-SNMP Agent ..	107
5.2.1 Modifications to the NET-SNMP Code	109
5.2.2 SNMP Agent Options	110
5.2.3 Main Storages Registration	111
5.2.4 SNMP Agent Daemon	112
5.2.5 RAgentX Agent Interface	115
5.2.6 Set Requests	119
5.3 RAgentX tools: Analysis and Design	121
5.3.1 Application Architecture	121
5.3.2 Main Classes	123
5.4 Execution Scenarios	124
5.5 Modifying JAX for the RAgentX Tools	131
5.6 Development and implementation environment	139
5.7 Limitations	139
5.8 Summary	143
6. Results and discussion	145
6.1 Introduction	145
6.2 Footprint	147
6.2.1 Management of a NE	147
6.2.2 Management from a NE	154
6.3 Bandwidth Use	159

6.3.1 Test Cases	159
6.3.2 Packet's Length Comparison	160
6.4 Processing Time	166
6.4.1 Test Configuration	167
6.4.2 Processing Time Comparison	169
6.5 Discussion.....	171
6.6 Summary.....	172
 7. Conclusions and Future Work	 173
7.1 Conclusions.....	173
7.2 Thesis contributions.....	174
7.3 Future Work.....	176
 References.....	 179
 Appendix A	 184

List of Figures

Figure 1.1 SNMP Architecture	3
Figure 1.2 Ubiquitous Network in a household environment.....	6
Figure 1.3 pdaMIB/ ARAX-SNMP agent Model	16
Figure 1.4 RAgentX tools/ ARAX-SNMP agent Model.....	16
Figure 2.1 J2ME software layer stack.....	30
Figure 3.1 AgentX Architecture	40
Figure 3.2 AgentX PDU Header	47
Figure 3.3 AgentXGetPDU format.....	49
Figure 3.4 Example of a SearchRange	51
Figure 3.5 Variable binding encoding.....	52
Figure 3.6 AgentXResponsePDU format	54
Figure 3.7 AgentXTestSetPDU format	56
Figure 3.8 Message flow between the AgentX master agent and subagent	58
Figure 3.9 SNMP Set request message flow.....	60
Figure 4.1: ARAX-SNMP architecture.....	68
Figure 4.2 pdaMIB/ARAX-SNMP Agent Model	69
Figure 4.3 AgentX subagent side Class Diagram.....	73
Figure 4.4 Code Segment from the PdaMIBDemo class (I)	76
Figure 4.5 Code Segment from the PdaMIBDemo class (II)	76
Figure 4.6 Code Segment from the AgentXConnection class	76
Figure 4.7 Code Segment from the AgentXReader class	77
Figure 4.8 Tree representation of a MIB table.....	81

Figure 4.9 SNMPGet command.....	87
Figure 4.10 SNMPTable command	90
Figure 4.11 Results from a snmptable request.....	90
Figure 4.12 Message flow between a master agent and the subagents when an ‘snmptable’ command is issued	92
Figure 5.1 RAgentX protocol. Messages flow.....	104
Figure 5.2 AgentX and SNMP Set phases.....	105
Figure 5.3 NET-SNMP Functional Architecture	108
Figure 5.4 SNMP agent daemon started as an AgentX master agent	110
Figure 5.5 SNMP agent daemon started as an AgentX subagent.....	111
Figure 5.6 Definition of an agent’s default store registration.....	112
Figure 5.7 Array with snmpd’s options	113
Figure 5.8 The variable that specifies if RAgentX is used is set to 1.....	114
Figure 5.9 SNMP daemon started as an RAgentX agent	114
Figure 5.10 The original “handle_master_agentx_packet” function.....	116
Figure 5.11 The modified “handle_master_agentx_packet” function	117
Figure 5.12 The “handle_agentx_packet” function	120
Figure 5.13 RAgentX tools/ ARAX-SNMP agent Model.....	122
Figure 5.14 Execution Scenario 1	125
Figure 5.15 RAgentXGetPDU requesting two variables	127
Figure 5.16 RAgentXResponsePDU answering a Get request.....	128
Figure 5.17 Response received from the ARAX-SNMP agent when an RAgentXGet request is issued.....	129

Figure 5.18 RAgentXSetPDU to set one variable.....	130
Figure 5.19 RAgentXResponsePDU answering a Set request	131
Figure 5.20 Result presented to the PDA user when an RAgentXSet request is issued	131
Figure 5.21 New AgentXOID constructor	132
Figure 5.22 Method “ifWithPrefix()” from class AgentXOID.....	133
Figure 5.23 Original method “getOIDBytes()” from the AgentXPDU class	134
Figure 5.24 New method “getOIDBytes()” from the AgentXPDU class.....	135
Figure 6.1 Test Configuration.....	168

List of Tables

Table 2.1 Comparison between different wireless technologies	28
Table 2.2 Comparison of J2ME tools and development toolkits	34
Table 3.1 Implementations of AgentX	62
Table 6.1 Agents/subagents Disk Space Occupation (DSO) Comparison	149
Table 6.2 Agents/subagents Total Amount of DSO Comparison	150
Table 6.3 Agents/subagents Footprint Comparison	151
Table 6.4 Agents/subagents Total Amount of Footprint Comparison	152
Table 6.5 Management Tools Disk Space Occupation Comparison	156
Table 6.6 Management Tools Total amount of DSO Comparison	156
Table 6.7 Management Tools Footprint Comparison	157
Table 6.8 Management Tools Total Amount of Footprint Comparison	157
Table 6.9 Test Case Length Comparison	160
Table 6.10 SNMP, AgentX, and RAgentX GET packets. Length comparison	161
Table 6.11 SNMP, AgentX, and RAgentX SET packets. Length comparison	162
Table 6.12 Processing Time Test Case	167
Table 6.13 Average Processing Time	169

CHAPTER 1

Introduction

This thesis investigates about the management of Ubiquitous Networks by developing: a) software that allows small devices, like PDAs, to be managed like any other NE in a network; b) tools that enable a small device to act as a manager, and c) a resource-constrained compatible protocol that enables the communication between resource-constrained devices acting as network managers and SNMP agents.

1.1 Background

Network management is the execution of the set of functions required for controlling, planning, allocating, deploying, coordinating, and monitoring the resources of a network. This includes performing functions such as configuration management, fault management, security management, performance management, and accounting management.

In any Network Management System (NMS), there are two basic elements: the network control center and the network nodes or network elements (NE) managed. The network control center is the entity that initiates requests for management information from managed systems or receives spontaneous management-related notifications from managed systems. The NEs managed are the resources in the network to be managed.

These resources are capable of sending and/or receiving management information over a communication channel.

The software needed for accomplishing the network management activities resides in the NE managed as well as in the network control center. The software that resides in each NE managed performs the following tasks [1]:

- Collects statistics on network-related activities.
- Stores statistics locally.
- Responds to commands from the network control center.
- Sends messages to the network control center when local conditions undergo significant changes.

Since manual management is prone to human errors, a NE can either support any of the standard Network Management (NM) protocols or create its own proprietary protocol. A NE should be managed for Fault, Accounting, Configuration, Performance and Security. Management protocols include SNMP, CMIP, TMN and many others. Choosing a protocol requires analyzing of NE characteristics, and NE's expected behavior.

SNMP and AgentX

In the SNMP world, there are managers and agents. The manager is the console or set of tools through which the network administrator performs network management functions. Agents are the entities that interface to or are part of the actual device being managed.

Every NE managed contains an SNMP agent. The SNMP architecture is depicted in Figure 1.1.

Figure 1.1 SNMP Architecture

Bridges, hubs, routers or network servers are examples of NEs managed that contain managed objects. These objects are arranged in what is known as a virtual information database, called a management information base or MIB.

Ordinary SNMP agents are monolithic. That is, they implement managed objects of all services and components in a single program. If the implementation of a MIB variable needs to be changed or a new variable wants to be added to the MIB, the whole agent has to be recompiled. The SNMP protocol does not provide any mechanism for **dynamically** extending the set of managed objects supported by a particular agent. This issue is addressed by the AgentX protocol.

The AgentX protocol is part of the SNMP family, and it defines a standardized framework for **dynamically** extending SNMP agents. In the AgentX framework there are:

- AgentX master agents: Entities that understand the SNMP protocol as well as the AgentX protocol, and accept registration of MIB regions (or portions of a MIB) by subagents.
- AgentX subagents: Entities that dynamically register MIB regions with the AgentX master agent, and perform management operations on MIB variables. They only understand the AgentX protocol.

The subagents allow us to dynamically extend the SNMP agent by dynamically registering the MIB they implement. From the manager's point of view, this extension of the SNMP agent is invisible; an extensible agent behaves exactly as would a non-extensible (monolithic) agent that has access to the same management information.

The SNMP architecture will be used as the base for the model developed in this thesis. A more detailed description of SNMP and AgentX is provided in Chapter 3.

Ubiquitous Networks

Networks and distributed processing systems are of growing importance and, indeed, have become critical in the business world. The trend is toward larger, more complex networks supporting more applications and more users. The term 'ubiquitous' is used

when computers and computation are everywhere, embedded in walls, tabletops, and other everyday objects [8].

As an example let us take the network formed in a house environment. In many houses nowadays, one may find several computers, laptops, etc. In addition there are many appliances with embedded computing systems: from alarm clocks to microwaves, fridges, dishwashers, etc. The network that these devices would form could be considered a Ubiquitous Network (UN). Three aspects that characterize these kinds of networks are:

- Resource limitation: Many of the devices in the network are everyday use appliances. These devices are resource-constrained.
- Wireless Communication: In order for these devices to be part of the UN, they need to be able to communicate with other devices. Most of everyday use devices are not built with the idea of communication in mind. One of the predominant ways of communication is wireless communication, although it is not the only one¹.
- Mobility: Some of these devices are mobile.

The goal of UN management is to manage the network these devices form. An example of a domestic UN is depicted in Figure 1.2.

A NMS should be able to monitor and control the different nodes in the network, independently of the characteristics of these NEs. In the case of a UN, the NMS should

¹ A more detailed list of communication technologies is given in Section 2.2.3

be able to manage a network formed by resource-constrained devices as well as those that are non resource-constrained.

Figure 1.2 Ubiquitous Network in a household environment

In this thesis, I apply the SNMP framework to a UN. For this, I set an SNMP agent as the core of the network management system and the AgentX subagents on each one of the UN devices to be managed. In the UN formed in the household environment in Figure 1.2, each one of the everyday appliances depicted would have an AgentX subagent installed. The subagents implement MIB variables and register them with the SNMP agent. The AgentX protocol allows the manager to access the information management the subagents implement.

This is not the only way the SNMP framework can be applied to a UN. A complete SNMP agent could be installed in each one of the devices to be managed. However this solution is not feasible for all the resource-constrained devices in a UN. A more detailed discussion is presented in Chapter 6.

PDAs (Personal Digital Assistants) are resource-constrained, mobile, and capable-of-communication devices (very often this communication is wireless). These are the same characteristics with which I have identified devices that might form UNs. For this reason, I will take PDAs as representative resource-constrained NEs of UNs, and I will use “PDAs” to refer to the resource-constrained devices to be managed in a UN. For extrapolation in this thesis, everything that can be applied to a PDA can be applied to any other resource-constrained device in a UN.

SNMP and AgentX for operational purposes

SNMP and AgentX are protocols designed for management purposes. However, they can also be used for operational purposes.

The network management system of a UN is in charge of the proper functioning of the network. Using SNMP and AgentX the manager of a UN can check the configuration of the devices, if they are working properly, the software installed in them, if this software needs to be upgraded, if any errors have occurred, etc. Furthermore, in a UN, the SNMP and AgentX protocols could be also used for operational purposes.

If SNMP and AgentX are used for operational purposes, these protocols can be utilized to cause the devices to execute some kind of action. For this a function that performs an action needs to be associated to the value of a variable. For example, let us consider as a device in a UN the light switch of a room. The MIB of this device implements a variable that represents if the light of the room is on. Let us call this variable “LightStatus”. This variable contains the number 1 if the light is on and 0 if the light is off. The “LightStatus” variable has associated a function that switches the light from on to off and vice versa. This function is invoked when the value of the “LightStatus” variable is changed. The manager can request the “LightStatus” variable (using an SNMPGet request) for management purposes. However, he can also turn the light of the room on and off by changing the value of the variable. If the variable contains the value 0 and the SNMP manager using an SNMPSet request changes it to 1, the “function” associated to the “LightStatus” variable is invoked and the light is turned on.

This is a very simple example of how SNMP and AgentX can be used for operational purposes. In this example I have used SNMP to switch on and off the light of a room. However, SNMP and AgentX can be utilized for operational purposes with any device in a UN.

1.2 Motivation

At the time of writing this thesis, there was no real solution to manage UNs available on the market. There were several proprietary solutions for the management of PDAs [3] [4] and even a program to control the cycles of a washing machine through the Internet using

the Web Ready Appliance Protocol (WRAP) [5]. But none of these solutions really dealt with the issue of managing a UN.

Taking into account the three characteristics of the devices that might form UNs listed in Section 1.1 (resource-constrained, wireless communicated and mobile), I state some important characteristics that the NMS should have for UNs:

- The software to be installed in the devices must be compatible with the resource-constrained features of these devices.
- I assume that all UN environments are wireless. For this reason, the protocol used for the communication should be lightweight (in processing time, use of bandwidth, and/or footprint).
- Because some of the devices are mobile, the NMS should provide a mechanism for the **dynamic** incorporation and withdrawal of devices into the network. Hence, it should allow the **dynamic extension** of the management information of the network that this mobility demands.
- The NMS should adopt a standard, because of the variety of NE vendors.

The UNs are still in development and the concept of UNs is very recent. For this reason, no methodology has yet been established for managing this kind of network.

Most routers, switches, and other devices that are usually NEs of a network, are fabricated with the idea of managing them in mind. Many of them come with built-in software or hardware needed to accomplish the network management activities. On the

other hand, many of the devices in a UN are items of everyday use. They do not come with management capabilities built in. There is a need to develop software that will allow NEs in a UN, like electronic appliances, to be managed.

The solution for the management of UNs should be standard rather than proprietary solutions. It should manage resource-constrained as well as non resource-constrained devices, independently of the vendor.

The NEs to be managed in a UN might not be active in the manager's range of action all the time. In addition, they might go from the range of action of one manager to another. This can be pictured very well when thinking of PDAs as mobile devices of a UN. A PDA can be carried by its user from one manager's range of action to another; can be turned on and off (in this case the PDA is not active), etc. When the PDA is off the manager cannot access the management information that the PDA contains. This can happen even to devices that are not mobile; they can be incorporated into the network at any time. In a house, for example, a fridge can be incorporated to or withdrawn from the network after the network has been initialized. The management information of the fridge should be incorporated into the general management information without causing the whole system to collapse, or without it needing to be reinitialized.

When working with SNMP I would like to be able to extend the MIB dynamically as the network elements come into the manager's range of action, or simply, as new devices (even static ones) are incorporated into the network. Each time a PDA, for example, goes

into an SNMP manager's range of action and registers its MIB with the SNMP agent, the manager has access to the MIB variables. This very real need to dynamically extend the management objects without having to stop to recompile the agent with the new MIB, and restart the SNMP agent has given rise to a variety of "extensible agents".

This dynamic registration can be accomplished, as mentioned in Section 1.1, by using the standard AgentX protocol. AgentX has been chosen in this thesis to be the protocol for the communication between the SNMP agent and the software running on the small device to be managed because AgentX is a resource-constrained compatible protocol. The lightness of the AgentX protocol will be discussed in Chapter 6.

I also want to be able to manage the network from a PDA or other small device. In order to do this the PDA should communicate with the SNMP agent, request information from it, and present it to the manager, the PDA user, in an intelligible way. Due to the characteristics of the UN the most critical factor of the resource-constrained devices, and for this my main focus, is the footprint. The protocol for the communication between the small device and the SNMP agent should be resource-constrained compatible. This protocol could be used instead of SNMP when the device used as a manager is a resource-constrained device.

1.3 Thesis goals

The ultimate goal of this thesis is the management of the many small devices in a UN, like one would manage any other NE with SNMP. For this, I need to implement a

management MIB whereby it can on request provide information to a manager and can accept management requests to modify certain values in the MIB for configuration or operation management purposes. Furthermore, I want to use a small mobile device as a mobile manager.

The specific goals for this thesis are to:

1. unify the management process of NEs in UNs, (so that it can be applied to resource-constrained as well as non resource-constrained devices). The solution should be compatible with standards like the SNMP family rather than using proprietary protocols;
2. develop the software needed for the management of resource-constrained devices;
3. solve the problem of having devices dynamically incorporated into and withdrawn from the network by making use of extensible agents;
4. make use of a resource-constrained compatible protocol for communication between the small devices of a UN to be managed and the SNMP agent;
5. develop and explore the use of a new footprint lightweight protocol that allows a small device, like a PDA, to dynamically become the manager of other NEs of the network. This footprint lightweight protocol could be used instead of SNMP when the device that acts as a manager is resource-constrained; and
6. develop the necessary tools that allow a PDA to act as a manager.

1.4 Thesis Contributions

To achieve these goals the pdaMIB application and the RAgentX (Reverse AgentX) tools have been developed, as well as the RAgentX protocol. The application and the tools run on PDAs. Taking into account the list of goals presented in Section 1.3, the contributions of this thesis are:

a) To unify the management process of NEs in UNs, I have used the AgentX protocol that is part of the SNMP family. The RAgentX protocol, based on the AgentX protocol, has also been developed and implemented for PDAs.

These protocols do not replace the SNMP protocol used by classic network management applications to interface with the SNMP agent services in the NEs; instead, they complement the management application/SNMP agent interaction model providing a resource-constrained compatible solution for the management of heterogeneous and dynamic networks.

b) The software developed for the management of resource-constrained devices is the pdaMIB application. This application is resource-constrained compatible. It has been developed using specialized tools for the implementation of software for small devices like PDAs. The pdaMIB application allows a manager to access the management information in the device.

The pdaMIB application and the RAgentX tools have been developed for PDAs because they represent small memory, small storage, small display, slow performance, slow network bandwidth, less functionality, and less user-friendly input method devices.

c) I have solved the problem of having devices dynamically joining and leaving the network by using the AgentX protocol. This protocol allows the dynamic extension of SNMP agents.

The AgentX protocol specifies a method for subagents to dynamically advertise to a SNMP agent (AgentX master agent enabled) the information for which they are willing to take responsibility, meaning the management information they implement and for which they can accept management requests from a manager. The pdaMIB application is an AgentX subagent implementation that allows the PDA in which it resides to dynamically join a network and be managed.

d) The resource-constrained compatible protocol used for the communication between the small devices of a UN to be managed and the SNMP agent is the AgentX protocol. A detailed discussion on the lightness of this protocol is provided in Chapter 6.

e) The resource-constrained compatible protocol developed that allows a small device, like a PDA, to dynamically become the manager of other NEs of the network is the RAgentX protocol. This protocol enables the communication between the device that acts as a manager and the ARAX-SNMP agent (AgentX and RAgentX enabled SNMP agent).

f) The tools developed for allowing a PDA to act as a manager are the RAgentX tools. These tools allow the PDA on which they are running to dynamically become the manager of other NEs of the network, that is, the user of the PDA has access to the management information stored in any other NE managed of the network.

Figures 1.3 and 1.4 show the applications/ARAX-SNMP agent interaction model. They depict the two roles that the small device can play: the NE managed role and manager role. In summary, in the presence of an ARAX-SNMP agent, a small device with a pdaMIB application running on it can be managed (using the AgentX protocol) as any other NE of the network. A small device can also manage other NEs in the network (using the RAgentX protocol) if it has the RAgentX tools running on it.

In summary, the contributions of the thesis are:

1. Analysis of possible solutions for management of Ubiquitous Networks.
2. Definition of a MIB and implementation of an AgentX subagent for PDAs.
3. Integration of the AgentX protocol in PDAs. For this I developed the pdaMIB application and modified JAX to fit small devices (see Sections 4.4 and 5.5)
4. Definition and implementation of the RAgentX protocol.
5. Development of RAgentX tools for PDAs.
6. Development of the ARAX-SNMP agent, which is an AgentX and RAgentX enabled SNMP agent.

Figure 1.3 pdaMIB/ ARAX-SNMP agent Model

Figure 1.4 RAgentX tools/ ARAX-SNMP agent Model

1.5 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides a review of the basic characteristics of UNs, as well as PDAs and other small devices. It also discusses how to program PDAs, the available tools, etc. A review of SNMP, AgentX, and JAX (Java AgentX) is provided in Chapter 3. In Chapter 4, the pdaMIB application is explained in detail. It describes how this application fits in the AgentX framework. Chapter 5 provides the design details of the developed RAgentX protocol, and discusses the main features of the RAgentX tools. Chapter 6 presents the results and a discussion of important issues related to the AgentX and RAgentX protocols: lightness and resource-constrained devices compatibility. Finally, in Chapter 7, conclusions and contributions of this thesis are summarized, followed by suggestions for related future work.

CHAPTER 2

Background Information

This chapter provides a background introduction to Ubiquitous Networks. It introduces the concept of Ubiquitous Networks, as well as some of the challenges that these kind of networks encounter. The UN developed in this thesis contains PDAs. I have taken PDAs as representative resource-constrained NEs of UNs. Accordingly, I present a list of the PDAs' features and some wireless technologies that enable the communication between embedded devices. It is also presented the options that Java provides for programming small devices, as well as the different toolkits (TK) that are available in the market to develop Java applications in connected embedded devices. Lastly, I introduce VisualAge Micro Edition, the toolkit I chose to develop the applications that run on small devices.

2.1 Ubiquitous Computing Systems

2.1.1 Definition of Ubiquitous Computing

The term Ubiquitous Computing (UC) was introduced by Weiser [8] [6] [7] [9] in 1988. According to Weiser, the aim of Ubiquitous Computing is to introduce a new way of utilizing computer capacity that makes computers disappear into the background. Ubiquitous Computing puts computing in the background of our lives, so that we do not have to focus on the tools we use but rather get on with the tasks we wish to accomplish. The highest ideal is to make a computer natural, that we use it without even thinking

about it. The term 'ubiquitous' is used because computers and computation will be everywhere, embedded in walls, in furniture, and in other everyday objects [8].

There is no standard definition for the term "Ubiquitous or Pervasive Computing". The majority of the bibliography found on this topic, [10], [18], [13], [14], [15], and [16], presents Ubiquitous and Pervasive Computing as two equivalent terms.

Pervasive Computing [15] is used to refer to the strongly emerging trend towards numerous, casually accessible, often invisible computing devices, frequently mobile or embedded in the environment, connected to an increasingly ubiquitous network structure. Ideally, these systems will be also able adapt to our behaviours, so that we would spend less time learning how to use and configure computers [14].

This is an emerging technology that is influenced by many research fields. It is largely influenced by research and development in the fields of mobile computing, distributed systems and artificial intelligence. But the research in these fields alone is not sufficient to arrive at pervasive computing. Advances in other research areas (such as building smaller devices and faster processors) are also critical to the success of Pervasive Computing.

Ubiquitous Computing devices have different sizes, each suited to a particular task. There are four features that can be distinguished in Ubiquitous Computing systems [11] [12]:

- These devices are meant to be practically invisible, and casually used. It means that the user interfaces must be simple, and transparent.
- The ability to modify the application behavior based on the knowledge of the context in which it is to be used. This concept is also referred to as “context awareness.”
- The ability to capture user experiences or habits for subsequent recall.
- The support for collaboration of groups of people is another key objective of ubiquitous computing applications. These collaborations may be carried out in real-time or asynchronously.

Even though a lot of work has been done to achieve this ideal world presented by Weiser, we are not quite there yet. A proliferation of devices has indeed occurred, with commonly used devices such as PDAs, pagers, smart cards, laptops, and wall-sized electronic whiteboards. In most cases however, these devices are far from being “invisible” or “natural” to use. In addition, the development and deployment of the necessary infrastructure to support mobile computation is still in development.

2.1.2 Challenges in UC Systems

Some of the challenges that a generic UC System faces are inherited from distributed systems and mobile computing. Here are some examples of these challenges [17], [13], [18]:

- Privacy: By observing and recording everything a user does these systems are potential personal security hazards. They could be misused to leak out our actions,

preferences, and locations to others. The information they record could be revealed to others now or in the future.

- **Invisibility:** As mentioned previously, one of the aims of UC is that the devices are “invisible”.
- **Human Interaction:** A device should detect when the user is trying to interact with it, and not with other users.
- **Interoperability:** Every device or software service must be explicitly written to understand every other type of device or software that it may encounter. This is a very important obstacle limiting impromptu interoperability.
- **Scalability:** The communication infrastructure should be such to allow many devices to interact with each other simultaneously.
- **Secure communications:** The communication infrastructure should provide encryption and authentication mechanisms.

2.1.3 Research Projects on UC Systems

Pervasive computing projects have emerged at major universities and in the private sector. Examples of university projects include Project Aura at Carnegie Mellon, Endeavour at UC Berkeley, Oxygen at MIT, and Portalano at Washington University. Private sector examples include work at AT&T Research in Cambridge, U.K., and at the IBM T.J. Watson Research Center in the US. A detailed list of universities and enterprises doing research on UC systems is provided in [19]. However, none of them addresses the issue of management of UNs.

2.2 Issues for Personal Digital Assistants (PDAs) and Other Small Devices

Since the management of Ubiquitous Networks described in Chapters 4 and Chapter 5 of this thesis is applied to PDAs, a list of some of the PDAs' features that are relevant to this endeavor is provided: memory and storage capacity, operating systems, and communication technologies.

2.2.1 Memory and Storage Capacity

Some PDAs have less than one MB of installed memory, but others can come shipped with as much as 128 MB. Generally, PDAs are provided with 16 MB of RAM and 4 MB of ROM. There are also some units with 64 MB of RAM and 32 MB of ROM like Compaq iPAQ H3835, HP Jornada 567, and CasioCassiopeiaE-200. ViewSonic ViewPad 100 SuperPDA has 128 MB of installed memory, and 32 MB Installed ROM. Some units have expansions slots and/or the ability to have their memory upgraded internally.

PDAs employ memory to store the operating system, standard applications and additional programs. It is difficult to compare storage capacity between models given their different operating systems.

For this thesis I have used two emulators, a Palm m505 handheld (16MB RAM, 8MB ROM, Palm OS 4.1, 33MHz processor speed), and a Palm IIIc handheld (8MB RAM, 8MB ROM, Palm OS 3.5, 20 MHz processor speed).

2.2.2 Operating systems

There are a number of different operating systems (OS) used for PDAs. Some of them are proprietary operating systems. The two main PDA operating systems are PalmOS from Palm and Windows CE from Microsoft. Both license their systems to other manufacturers. The third major system, popular in Europe, is EPOC from Symbian.

PalmOS is the OS for all Palm models, certain IBM models, the TRGpro, and the Handspring Visor. The latest version of PalmOS is 5.0. Palm OS is simple to use and inexpensive.

Microsoft released Windows CE (WinCE) for small devices like PDAs and set-top TV controllers. WinCE is used as the OS on palm devices from manufacturers like Hewlett-Packard, NEC, and Casio. The newer versions of WinCE are now referred to as Pocket PC. The PocketPC OS is basically WinCE with more added features.

Compared to WinCE devices, Palm devices generally have slower processors and less RAM. This is due to the fact that WinCE consumes valuable memory (it uses memory from the system to run the operating system itself), and requires a faster processor in order to be as efficient as other operating systems, running slower processors, such as EPOC or Palm OS.

I chose a PalmOS device for the development of this thesis because the characteristics of PalmOS devices resemble the limitations of the resource-constrained devices in UNs.

2.2.3 Communication

As mentioned before, one of the main characteristics of ubiquitous devices is their capacity to communicate with the other devices of the system. Some devices will already be connected forming a wired-network, but most communicate wirelessly.

Wireless networks are the bridges between mobile users and wired networks. This communication is either done by pulsing infrared lights or modulating radio waves. There are various types of wireless networks. They can be classified according to the range they cover and the role they have [20], [21]: wide-area cellular networks, wireless LANs (with radio ranges typically from 15 meters to over 300 meters), wireless Personal Area Networks (PANs) (small office or home within 5-15 meter distances), and satellite networks. I will not comment on all of them, but simply focus on the most relevant ones for this thesis.

IEEE 802.11b

An example of a wireless technology used in wireless LANs is IEEE 802.11b [22]. IEEE 802.11b is a standard wireless Ethernet networking technology for both business and home. It uses the Ethernet protocol and CSMA/CA (Carrier sense Multiple Access with Collision Avoidance) for path sharing.

Any network adapter coming within range of another IEEE 802.11b network adapter or access point can instantly connect and join the network unless WEP (Wireless Encryption Protocol) is enabled.

IEEE 802.11b is a half duplex protocol. It uses the same 2.4 GHz range as many cordless phones so there exists the possibility of interference. A possible solution for this is to use 900 MHz cordless phones if using IEEE 802.11b in the same area. Next-generation IEEE 802.11b is faster, and operates in the relatively vacant 5 GHz band.

Xircom manufactures an IEEE 802.11b sled for the Palm m500 series and m125, as well as the SpringPort wireless Ethernet Springboard module for the Handspring Visor line of PDAs. Socket Communications manufactures a "Low Power Wireless LAN type I CF" card for Pocket PC and WinCE devices. This 802.11b wireless Ethernet card is called "Low Power" because it consumes about 1/2 power as any other wireless network card. These are two examples of manufacturers of IEEE 802.11b cards, but there are many more.

IrDA (Infrared Data Association)

Examples of wireless PAN technologies are IrDA and Bluetooth. IrDA [23] is an industry-sponsored organization set up to create international standards for the hardware and software used in infrared (IR) communication links.

The IrDA specifications call for two types of infrared communications, namely SIR (Standard IR) and FIR (Fast IR). The transmission speed of SIR is 115 Kbps. Fast IR is capable of delivering substantially higher transmission speeds. FIR is being used on some platforms to implement local area networking connectivity because it is capable of

throughput of up to 4 Mbps. To put this in perspective, a normal Ethernet network is capable of either 10 or 100 Mbps, or even 1Gbps.

Although the IrDA standard only specifies a connection from zero to one meter, many IrDA-compliant products can connect at distances greater than one meter. IrDA is sensitive to fog and other atmospheric conditions. These two characteristics make IrDA more suitable for wireless Personal Area Networks (small office or home.)

The majority of PDAs come with an infrared port, and many new phones are equipped with built-in IR capability as well. Some examples of devices that come with a built-in IrDA infrared port are Palm devices Palm III and later models, Sony CLIE PEG-T615C/S, Hewlett-Packard Jornada 568, etc.

Bluetooth (BT)

Bluetooth [24] is a short-range wireless technology that supports data, voice and content-centric applications. Bluetooth requires the inclusion of a low-cost transceiver chip in each device. The transceiver transmits and receives in a previously unused frequency band of 2.4 GHz that is available globally (with some variation of bandwidth in different countries). Connections can be point-to-point or multipoint. The maximum range is 10 meters. Data can be exchanged full duplex at a rate of 1 Mbps (up to 2 Mbps in the second generation of the technology). A frequency hop scheme allows devices to communicate even in areas with a great deal of electromagnetic interference. Built-in encryption and verification is provided.

Right now there are cell phones, PC cards, CompactFlash cards, printer modules, access points (that serve and route Internet connections to BT clients like PDAs), USB adapters, the HP 995c printer, headsets, and some notebooks from Sony and IBM with Bluetooth built-in capability.

Discussion

Bluetooth and IEEE 802.11b are different but complementary technologies. IEEE 802.11b is largely applied to LAN access, while Bluetooth main focus is PANs. Both are methods that allow computers to communicate to other devices, both use wireless technology, and both operate in the 2.4 GHz spread spectrum band.

PDA hardware manufacturers that have investigated supporting both standards concluded that while IEEE 802.11b is the preferable wireless LAN technology in most scenarios, its greater thirst for battery power and higher component cost compared to Bluetooth has to be taken into account. For these reasons (especially the battery life issue), Bluetooth can be a better wireless LAN choice for handhelds. Speed is an issue, though: Bluetooth is very slow when compared to wireless Ethernet. Both have new, faster versions coming out, but the speed differential will be even more pronounced in wireless Ethernet's favor.

Bluetooth's biggest potential use is cable replacement. Bluetooth should also eliminate the proprietary adapters found at the ends of the many cables used to attach devices to one another. Bluetooth is well suited for devices; IEEE 802.11 is, in turn, designed

specifically for driving WLANs. This is the reason why IEEE 802.11b was chosen for the development of this thesis.

The Table 2.1 shows a comparison between the three discussed technologies, suitable for Ubiquitous Networks.

Technology	Speed	Maximum Distance	Type of Network
IrDA FIR	4Mbps	1 meter	PAN
IrDA SIR	115Kbps	1 meter	PAN
Bluetooth	1 Mbps	10 meters	PAN
IEEE 802.11b	4 Mbps	100 meters	WLAN

Table 2.1 Comparison between different wireless technologies

2.3 Programming Small Devices

By “small devices” I refer to resource-constrained devices that have any kind of computing constraint, mainly constraints regarding storage capacity, speed, power consumption, communication bandwidth and latency.

Even though not all ubiquitous devices are resource-constrained in terms of memory or CPU, I will consider the worst-case scenario. In this way it is assured that I can program

all kinds of devices, from the feature poor ones to those with more resources. For the same reason, I will also assume that the communication between the devices of a UN is wireless.

In the following subsections I will discuss the options that Java provides for programming small devices, as well as the different toolkits available in the market for it.

2.3.1 Java for Small Devices

Sun has regrouped the Java technology into three editions taking into account the different environments and products to be developed: J2EE (Java 2 Enterprise Edition), J2SE (Java 2 Standard Edition) and J2ME (Java 2 Micro Edition). For this thesis J2ME is the most relevant of the three because it specifically addresses small devices, from smart cards to set-top boxes.

J2ME [25] is a Java-based programming language that facilitates building client-side applications optimized for mobile devices that have limited resources. J2ME is a set of specifications targeted at mobile information devices like PDAs, cell phones, two-way pagers, set-top boxes, etc. Until recently, these devices were usually programmed in C, C++, or an assembly language. Java is a good alternative for two reasons: Java code is portable, (although the constraints of small devices mean that most of this benefit is erased in J2ME), and Java's promise of "Write Once, Run Anywhere" is true, to a degree, in the J2ME world.

As shown in Figure 2.1, the J2ME architecture is composed of three modular and scalable layers of software built upon the Host Operating System of the device: Java Virtual Machine, Configurations and Profiles [25].

Figure 2.1 J2ME software layer stack

Java Virtual Machine (JVM)

This layer is an implementation of a Java virtual machine that is customized for a particular device's host operating system and supports a particular J2ME configuration. In the case of PDAs, the configuration used is CLDC. For the CLDC configuration the JVM is called KVM or Kilobyte Virtual Machine. A more detailed explanation of the different configurations is given in the next subsection.

The goal for the KVM technology [25] was to create the smallest possible "complete" Java virtual machine that would run in devices with limited resources, but maintaining all the central aspects of the Java programming language. Three are the challenges that the KVM tries to surmount. These challenges are decreasing the footprint of the virtual

machine and class libraries themselves, decreasing the memory used by the virtual machine during execution, and permitting components of the virtual machine to be configured to fit the features of particular devices.

The KVM is in the range of 40 to 80 Kbytes (depending on compilation options and the target platform). It is suitable for devices with as little as 160 Kbytes of total memory available (128 Kbytes for the storage of the actual virtual machine and libraries).

A relatively small portion of the KVM code is platform dependant. This code is isolated in a few files. Porting the KVM to a specific device demands modifying only these files.

Configurations

This layer provides the minimum set of JVM features and Java class libraries available on a particular category of devices. Currently, there are two standard J2ME configurations: CDC (Connected Device Configuration) and CLDC (Connected, Limited Device Configuration) [25] [26]. The CDC configuration is a superset of the CLDC.

CDC configuration targets shared, fixed and always connected information devices, that are resource poor (few megabytes of available memory). CDC uses a full-featured Java virtual machine.

CLDC devices use KVM. Some examples of this kind of device are PDAs, cell phones, two-way pagers, etc. Devices targeted by CLDC are characterized generally by: 160 to

512 kilobytes of total memory (including both RAM and flash or ROM), limited power (often battery powered operation), connectivity with a wireless network (9600 bps or less bandwidth), poor or no user interfaces, and 16- or 32-bit processor.

Profiles

This layer provides the minimum set of Java technology-based APIs available on a particular family of devices [25] [26]. A device can support multiple profiles. Profiles are built upon a particular configuration. Applications are written for specific profiles, and can run in any other device that supports the same profile. Two are the purposes of the Profile layer: device specialization and device portability.

The only profile currently developed for CLDC configuration is the Mobile Information Device Profile (MIDP). The Java Community Process (JCP) is developing a new CLDC profile specification that would fit the specific needs of PDAs. However, it has been so long in the making that it might be too limited for current and future PDAs.

Although the goal is to get applications fully portable between devices of the same kind, there is a potential difficulty in porting MIDP applications (or MIDlets) across different MIDP devices. Manufacturers can add to the MIDP (in the form of an API) their own OEM's (Original Equipment Manufacturer) "extravagant features". This is contrary to the portability spirit of Java. It will be up to the developers to choose between specializing their applications for a specific device or the ease of porting their MIDlets across multiple MIDP devices.

HTTP is the only protocol a MIDP implementation must support, all other protocols (such as TCP, Transmission Control Protocol, or UDP, User Datagram Protocol) are optional.

2.3.2 J2ME Development Toolkits

There are many different development toolkits on the market for the development and deployment of Java applications on connected embedded devices, and there are also many features that they provide. Some of the features are: device emulators, device editor, pre-verification tools², jad/jar file packaging³, debugging, etc.

Some features are more important than others. For example, a simple KVM implementation without system class pre-linking support requires more volatile memory than a KVM implementation with system classes (or even applications) preloaded into the device.

The Table 2.2 below compares different J2ME Development Toolkits [26].

The toolkit used for the development of the applications described in Chapters 4 and Chapter 5 is VisualAge from IBM [27] [28]. VisualAge Micro Edition (VAME) is an Integrated Development Environment (IDE) with a set of tools and runtime components. These tools and components allow the development of MIDP/CLDC compatible

² The desktop compiler verifies that the compiled code can run with J2ME's KVM.

³ A jar file is used to group all the classes together so that the application is easy to distribute and deploy. The Java Application Descriptor, jad, provides information about the contents of the jar file

applications to be run on connected embedded devices. VAME provides full access to the Palm OS API.

Toolkit	Device Emulator	MIDlet Packager	Debugger	Pre-verifier	Source Editor
CodeWarrior for Java	Yes	Yes	Yes	Yes	Yes
Zucotto Whiteboard	Yes	Yes	Yes	Yes	Yes
Forte for Java	No	No	Yes	No	Yes
JBuilder	No	Yes	Yes	No	Yes
VisualAge Micro Ed.	No	No	Yes	No	Yes
Sun Wireless	Yes	Yes	No	Yes	No
Nokia J2ME	Yes	Yes	No	Yes	No
Siemens J2ME	Yes	Yes	Yes	Yes	No
RIM BlackBerry JDE	Yes	NA	Yes	Yes	Yes

Table 2.2 Comparison of J2ME tools and development toolkits

IBM has its own virtual machine, called J9. IBM J9 is highly portable. It supports several operating systems, such as Palm OS, Linux, and WinCE. It is also portable across a range of processors Intel x86, Motorola 68K, ARM and StrongARM, MIPS, PowerPC, Hitachi SuperH SH-3 and SH-4.

VisualAge Micro Edition provides two base class libraries that have been custom-designed for Palm OS:

- **jclPalmCldc**: A full beta implementation of the CLDC specification, for portability of applications between various CLDC-enabled platforms.

- jclPalmXtr: Derived from VisualAge Micro Edition's Extreme (jclXtr) class library, for an even smaller footprint.

Once Java source files are compiled into class files, the VisualAge Micro Edition SmartLinker removes methods and classes that are included in the project but are not actually used, creating the smallest possible executable file. The SmartLinker creates one or more JXE (J9 eXExecutable) files, which run efficiently on the J9 VM. The use of JXE files minimizes start up time, and footprint.

VisualAge Micro Edition's support for Palm OS uses PilRC; a standard Palm resource compiler that translates Palm OS textual resource files into compact binary resources for building user interfaces. The jclPalmXtr library provides support for the HTTP and the TCP protocols.

CHAPTER 3

Background Information on SNMP and AgentX Protocols

The AgentX protocol allows subagents to dynamically register their MIB regions without interrupting the activity of the agent they register with. AgentX was developed primarily as a protocol for inter-process communication between a master agent and its subagents running on the same machine. However, it can also be used over a wired or wireless network. In this thesis, I present the AgentX protocol as a possible solution for the management of small devices in a UN. Using AgentX the small devices of a UN, acting as AgentX subagents, can dynamically register the MIB regions they implement.

3.1 The SNMP Protocol

SNMP management, often called Internet management, is the protocol most widely used in NMS [2]. This protocol is capable of managing more than just the Internet since it is implemented in most network components in an enterprise network system.

As mentioned in Section 1.1 and depicted in Figure 1.1, in the SNMP framework there are managers and agents. SNMP [1] is the protocol to transfer messages between these two entities. For the management of the NEs of a network what is actually transferred between the SNMP agent and the manager, using the SNMP protocol, is the information contained in the MIBs. A basic MIB is defined as part of the SNMP framework. Product

developers can extend this basic MIB by creating/implementing and registering new MIBs.

MIB's Object Identifiers

The MIB variables or objects are accessed via their Object Identifier, or OID. The OIDs are defined using the mechanisms provided by SMI (Structure of Management Information). The name space for MIB OIDs is hierarchical and can be seen as a tree. An OID in the hierarchy is written as a sequence of sub-identifiers beginning at the root and ending at the object. By tracing the path from the root of the tree to the leaves (traversing the tree), you can find the unique OID of each MIB variable.

Each variable of a MIB has a textual and a numeric representation. For example, the textual representation of the variable that contains the contact information of a NE is:

iso.org.dod.internet.mgmt.mib-2.system.sysContact.0

while the numeric OID or numeric representation is:

.1.3.6.1.2.1.1.4.0

A “MIB region” is a portion or subtree of a MIB tree. It is a group of leaves (OIDs) as well as the parents of these leaves. A MIB region is usually specified by indicating the ‘root’ (or top) of the subtree.

There are in the market MIBs (or more accurately, MIB extensions) for each set of related network entities that can be managed. In order to extend the basic MIB the new MIB implemented by a developer needs to be registered with the SNMP manager (either directly, or via AgentX). When a new agent is added to extend the manager's domain the manager must be provided with the new MIB component that defines the manageable features of the resources managed through that new agent. After a MIB component has been registered a manager can request the variables of this MIB specifying its OIDs.

SNMP and AgentX messages never contain the textual representation of a variable. Variables are always requested using their numeric OID (or numeric representation). SNMP tools have access to files that store the MIB definitions. The MIB definitions provide the relationship between the textual and numeric representations of MIB variables. If a manager requests a variable given the textual representation, the SNMP tools translate the name into the numeric OID (accessing the MIB definitions), and issue the request.

Most of the SNMP tools can work without any MIB files at all. The MIB files are only used for translating between numeric and textual forms for queries and responses.

Context

In SNMP an Access Control Model (ACM) defines mechanisms to determine whether access to a managed object should be allowed.

An SNMP context is a set of managed objects accessible by an SNMP entity. A managed object may exist in more than one context, and an SNMP entity potentially has access to many contexts. For the authentication of managers and agents SNMP v1 uses community names. These community names are “passwords” that the managers send with the requests in order to be authenticated. The agent uses the community string to determine the manager's access privileges.

SNMPv3 supports authentication mechanisms such as SHA, MD5, and DES encryption. SNMPv3 has three levels of security: The lowest level does not provide authentication or privacy (noAuthNoPriv). This level's security is thus comparable to SNMPv1. The second level provides authentication but no privacy (AuthNoPriv), and the highest level provides authentication and privacy (AuthPriv).

3.2 The AgentX Protocol

3.2.1 Introduction

AgentX [29] [30] is a standard protocol that overcomes a very real need to dynamically extend the SNMP agent's MIB. The protocol allows having a single SNMP agent and several subagents that can connect and register dynamically various MIB regions (without having to interrupt the management service). This way the whole management system can be more flexible, each subagent can be closer to the managed information, the SNMP agent's MIB can be extended dynamically, and the AgentX enabled SNMP agent needs to start only once.

AgentX is designed to be independent of any particular SNMP version. An AgentX SNMP framework consists of two types of entities: a master agent and one or more subagents. As mentioned, AgentX was originally elaborated as a protocol for the communication of processes (master agent and subagents) running on the same machine. The two entities communicate with each other either over a) TCP or b) UNIX domain sockets. The master agent is the intermediary between the SNMP manager and the AgentX subagent and understands both protocols: AgentX and SNMP. Figure 3.1 represents the two components (master agent and subagents) and their relationship with other elements of the SNMP/AgentX architecture.

Figure 3.1 AgentX Architecture

The AgentX protocol determines a method for subagents to connect to a master agent to register the information for which they are intended to take responsibility, that is, the

management information they implement and for which they can accept management requests from a manager.

It is the master agent's responsibility to maintain a table of the MIB regions the subagents have registered. When the AgentX master agent receives a SNMP request it finds in the table the subagent(s) responsible for the requested MIB region and dispatches the appropriate AgentX requests. As far as the core of the master agent is concerned, it does not really know whether a particular request has been passed off to a subagent or not - it calls the routine that is registered for a particular OID and waits for the answer. If that MIB region is actually handled by an AgentX subagent then the routine called is simply a "stub" that passes the request off to the subagent and waits for the answer. But for the master agent this process is invisible - it just looks like yet another MIB implementation routine.

The master agent itself does not have to contain any management information with the exception of information regarding currently connected agents and allocated MIB regions. The master agent is what would traditionally be thought of as the SNMP agent. It contains the agent protocol engine (SNMPv1, SNMPv2c, and/or SNMPv3). The master agent is in charge of the authentication, authorization, access control, and privacy mechanisms if SNMPv3 is used.

The AgentX subagents are responsible for providing access to management information. A subagent is essentially a small program that provides access to the MIB it implements.

A typical subagent when it starts, it contacts the master agent and registers the various MIB regions for which it has information. After, it runs in a loop and awaits requests from the master agent whereupon it gathers the information requested and returns the response. Finally, it unregisters the MIB region and closes the connection with the master agent. A subagent may do more than simply respond to messages (e.g. it can send traps), but all within the scope of the MIB it implements.

Index Allocation and Overlapping Subtrees

Subagents are independent processes that register their MIB regions without taking into account the actions of the other subagents. The AgentX protocol was designed to hide subagents from the existence of other subagents, and to make the access to these subagents transparent to managers. For these reasons, the master agent acts as mediator between the manager and the subagents (by providing a means with which to dispatch SNMP requests to subagents, as explained above) and arbitrates all conflicts between subagents.

To solve the conflicts between subagents the master agent provides an index allocation service and resolves overlaps in MIB region registrations in a deterministic form.

The overlapping conflict occurs when a subagent requests to register a MIB region that has already been registered. The MIB developed in this thesis does not present the problem of overlapping subtrees because, as it is explained in Section 4.1.4, each AgentX subagent registers a different row of a MIB table. Even though several subagents are

implementing the same MIB table, this is not considered overlapping because each subagent implements a different row of that MIB table. More details on overlapping can be found in [29].

Index allocation is a service that provides generic support for sharing MIB conceptual tables among subagents who are assumed to have no knowledge of each other. It allows several subagents to register different rows of the same MIB table. The master agent maintains a database of index objects (OIDs) and, for each index, the value that has been allocated for it.

The process followed to register a row of a table when using index allocation would be:

1. The subagent contacts the master agent for index allocation;
2. The master agent looks for a value not allocated yet, consulting the database of index objects, and returns an unassigned value to the subagent; and
3. The subagent registers the row using the value given by the master agent.

The purpose of index allocation is to prevent two subagents from trying to handle the same row of a table. If there is no intrinsic meaning to the table index values (i.e. they are arbitrary integers) then typically both subagents will start at 1. Both will request to register the first row. This will bring the two of them in to conflict over which one will get to use which index. That is where index allocation comes in - the master agent can stop the “wrestling” and hand out the index values fairly and evenly. If there is an intrinsic meaning in the index value, then as long as the MIB has been properly designed,

there should not be any contention over which row each subagent will register. In this case there is no need to use index allocation ("I live at number 28, so I will handle row 28. You live at number 26, and own 30-76 inclusive, so you can handle those rows").

To avoid conflicts when several subagents want to register rows in the same MIB table, AgentX provides two solutions:

- If the index to be allocated does not have an intrinsic meaning, the subagent should first request an index to the master agent, and then use it to register the row in the table, as explained above. For this the AgentX master agent needs to support index allocation.
- If the index to be allocated has an intrinsic meaning, the subagent can use it directly to register the row in the table. In this case there is no need for the master agent to support index allocation.

To summarize the characteristics of AgentX, I present a list of various advantages to using this protocol [30] [31]:

- AgentX is a standardized protocol and it is part of the SNMP family. Any programming language can be used to implement a MIB. AgentX master and subagent processes can be located and replaced arbitrarily.
- The master agent does all security and access control processing. The subagents do not have to handle these error-prone details.

- The AgentX protocol uses its own simplified encoding. There is no duplicated BER (Basic Encoding Rules) encoding/decoding at the master agent and the subagent.
- AgentX subagents are independent of the AgentX master agent's SNMP version.

There is a strong tendency to keep the existing SNMP APIs (Application Programming Interfaces) unchanged so that existing instrumentations can be turned into AgentX subagents without changes to the source code. A good example of this is the approach taken by the NET-SNMP project [32] to support AgentX. This AgentX enabled SNMP implementation is the one used in this thesis.

3.2.2 AgentX Protocol Data Units (PDUs)

AgentX PDUs consist of a common header followed by PDU-type dependant fields of variable length. Unlike SNMP PDUs, AgentX PDUs are not encoded using BER; instead they are transmitted as a continuous byte stream.

The PDU types available in AgentX are:

- The set of PDUs for “administrative” purposes. These are: (I have grouped them according to their function)
 - AgentXOpenPDU and AgentXClosePDU
 - AgentXRegisterPDU and AgentXUnregisterPDU
 - AgentXAddAgentCapsPDU and AgentXRemoveAgentCapsPDU
 - AgentXIndexAllocatePDU and AgentXIndexDeallocatePDU

- The set of PDUs types for “SNMP requests processing” are: AgentXGetPDU, AgentXGetNextPDU, AgentXBulkPDU, AgentXTestSetPDU, AgentXCommitPDU, AgentXUndoSetPDU, and AgentXCleanUpSetPDU.
- The AgentXResponsePDU is used to respond to the other PDUs.
- The AgentXNotifyPDU is used to send traps and AgentXPingPDU is used by the AgentX subagent to monitor the ability of the master agent to receive PDUs.

The structure of the PDUs varies depending on the PDU type. I will not describe all of them in this thesis. A very detailed description can be found in [29]. I will comment on the most representative ones for this thesis: AgentXGetPDU, AgentXResponsePDU, and AgentXTestSetPDU.

AgentX PDU Header

The AgentX PDU header is a fixed-format, 20-octet structure [29]. Figure 3.2 shows this structure. (The “h.” notation stands for “header”).

The AgentX header contains the following fields:

- h.version: The version of the AgentX protocol.
- h.type: This field specifies the PDU type. The most relevant AgentX PDUs’ types are: AgentXOpenPDU, AgentXClosePDU, AgentXRegisterPDU, AgentXUnregisterPDU, AgentXGetPDU, AgentXTestSetPDU, and AgentXResponsePDU.

- **h.flags:** This field is a bitmask to indicate network byte order, instance registration (used when registering MIB regions), non default context (used to specified that the context is not the one by default), and new or any index (used when it is needed to allocate an index, more details about index allocation were given in Section 3.2.1).

Figure 3.2 AgentX PDU Header

- **h.sessionID:** An AgentX subagent can establish more than one session. The sessions are opened by sending an AgentXOpenPDU to the AgentX master agent.
- **h.transactionID:** For each SNMP request sent by the manager to the AgentX master agent there is one or more AgentX PDUs associated to it. These PDUs are sent by the

AgentX master agent to the subagent. The transactionID uniquely represents the SNMP request and all the AgentX PDUs associated with it will have this transactionID in their header.

- h.packetID: Each AgentX packet generated has a different packetID. However, AgentXResponsePDU should have the same packetID as the received AgentX PDU for which it is the response.
- h.payload_length: The size in bytes of the PDU contents, excluding the header.

AgentXGetPDU

The AgentXGetPDU is sent by the AgentX master agent to one or more subagents whenever an SNMPGet request is made by the manager. The AgentXGetPDU's fields are depicted in Figure 3.3. (The "g." notation stands for "get request").

The AgentX PDU Header is part of the AgentX PDU, for this reason it has been depicted again in Figure 3.3 with the rest of the fields that form the AgentXGetPDU.

The AgentXGetPDU contains the following fields:

- g.context: Optional octet string field used when a non-default context is required.
- g.sr: This field is a SearchRangeList. It contains the requested variables for this session. A SearchRangeList is a contiguous list of SearchRange. Each SearchRange consists of two OIDs that are used to identify the requested variables. In AgentXGetPDUs the second OID is always null-valued.

Figure 3.3 AgentXGetPDU format

An OID is encoded as a 4-byte header and a variable number of 4-byte fields that contain the OID's sub-identifiers. The first field of the header, "n_subid", contains the number of sub-identifier fields that will follow the header. The second field of the header, "prefix", is used to reduce the length of the object identifier encodings. When this field contains a non-zero value "x", it indicates that the prefix "internet" (1.3.6.1.4) is used, and that the number "x" is the first sub-identifier after the prefix. For example, if the OID to be encoded was "1.3.6.1.4.1.2000.1.1.1.1.234", the field "prefix" would have the value "4" because the encoding of this OID uses the prefix "internet" and the sub-identifier that follows is 4. In this case, the value of the field "n_subid" would be "7" because there would be 7 sub-identifier fields following the header. This OID encoding is depicted in Figure 3.4 (as part of a SearchRange).

In a SearchRange, the first OID specifies the beginning of the range, while the second specifies the end of the range. The second OID is never included in the range. The first one is included depending on the value of the field "include" (if the field contains a 0 the OID is not included, if it contains a 1 the OID is included.) In AgentXGetPDUs the second OID of the SearchRange is always null-valued and the first one is always included because with this type of PDUs only one variable is requested.

Let us look at an example of a search range. To indicate the search range from "1.3.6.1.4.1.2000.1.1.1.1.234" (inclusive) to "1.3.6.1.4.1.2000.1.1.1.6.234" (exclusive) the SearchRange encoding would be as depicted in Figure 3.4.

Figure 3.4 Example of a SearchRange

AgentXResponsePDU

The AgentXResponsePDU consists of the common header plus 3 fields specific to this type of PDUS: res.sysUpTime, res.error, and res.index. (The “res.” notation stands for “response”).

It may also contain a list of variable bindings (VarBinds). The representation of VarBinds has two 2-byte fields (*v.type* and 2 bytes reserved for future use), a name (the OID), and the actual value data. The VarBind structure is depicted in Figure 3.5 (the “v.” notation stands for “VarBind”). This structure can also be seen in Figure 3.6 as part of the AgentXResponsePDU.

Figure 3.5 Variable binding encoding

The VarBind fields are:

- *v.type*: Indicates the variable binding’s syntax. Some of the possible values are: Integer (2), Octet String (4), and Object Identifier (6).
- *v.name*: This is the OID that names the variable.

- **v.data:** The actual value is encoded using a variable number of bytes, depending on the type of the variable indicated in **v.type**. For example, integers are encoded as 4 contiguous bytes. **OID** are encoded as shown in Figure 3.4. An octet string is represented by a contiguous series of bytes, beginning with a 4-byte integer whose value is the number of octets in the octet string, followed by the contained octets. An example of octet string encoding is **g.context**, shown in Figure 3.3.

The specific fields of **AgentXResponsePDU** are:

- **res.sysUpTime:** This field contains the current value of **sysUpTime** for the context indicated within the PDU to which this PDU is a response.
- **res.error:** Indicates error status. This field has 2 bytes.
- **res.index:** If there was an error this field indicates which variable binding failed within a received request PDU.

The **AgentXResponsePDU** is used in the communication between the **AgentX** master agent and an **AgentX** subagent always as a response to other PDU. As mentioned above, a **VarBind** list may follow the **res.index** field. The **AgentXResponsePDU** will contain a **VarBind** list only when responding to an **AgentXGetPDU**, **AgentXGetNextPDU**, or **AgentXGetBulkPDU**. The **VarBind** list contains the variables requested within the PDU to which this PDU is a response, and their values. Figure 3.6 depicts an **AgentXResponsePDU** with a **VarBind** list.

Figure 3.6 AgentXResponsePDU format

The only difference between the AgentXGetPDUs (depicted in Figure 3.3) and the AgentXResponsePDUs (depicted in Figure 3.6) is that the latter have a VarBind list and a group of fields to specify if an error has occurred, while the first ones have a

SearchRange list and optional string field used when a non-default context is required. The differences have been highlighted.

AgentXSet PDUs

AgentX has four PDUs for processing a Set request: AgentXTestSetPDU, AgentXCommitSetPDU, AgentXUndoSetPDU, and AgentXCleanUpSetPDU.

An AgentXTestSetPDU consist of an AgentX header, an optional non-default context (s.context), and a VarBind list containing the requested VarBinds. The v.name of each VarBind contains the numeric OID of the variable that is requested to be set, and each v.data contains the value to which the variable is being set.

Most SNMP request PDUs are mapped to similar AgentX PDUs: Get, GetNext, etc. But the AgentX protocol does not use one PDU for setting MIB variables. Instead, it uses four different PDUs. Each one of them corresponds to a different phase into which the process of setting MIB variables is divided. These phases are: TestSet, CommitSet, and UndoSet or CleanUpSet. Accordingly, the PDUs are: AgentXTestSetPDU, AgentXCommitSetPDU, AgentXUndoSetPDU, and AgentXCleanUpSetPDU. The latter three PDUs consist of the AgentX header only. The only difference between the AgentXGetPDUs (depicted in Figure 3.3) and the AgentXTestSetPDUs (depicted in Figure 3.7) is that the latter have a VarBind list, while the first ones have a SearchRange list. In Figure 3.7 the differences have been highlighted.

Figure 3.7 AgentXTestSetPDU format

3.2.3 Messages Flow

The basic flow of messages between the AgentX subagent (residing in the PDA), and the AgentX master agent (ARAX-SNMP agent) is depicted in Figure 3.8. This flow of messages is based on the AgentX protocol.

It can be seen from Figure 3.8 that there are three types of messages corresponding to the three phases of communication between the subagent and the SNMP agent: a) session opening; b) SNMP request processing; and c) session closing.

a) Session opening: These messages are related to the opening of a session. They initiate AgentX sessions between the subagent and the master agent, and register the MIB region that the subagent implements.

- AgentXOpenPDU: Contains the information needed to open the connection. The SessionID field is blank because it will be filled in by the master agent.
- AgentXResponsePDU: The first AgentXResponsePDU returns a SessionID to the subagent. From this moment on all the PDUs between the master agent and the subagent will have this SessionID in their header. The second response confirms that the MIB region has been registered by specifying the first and last OID of the region registered.
- AgentXRegisterPDU: With this PDU the subagent specifies to the master agent the MIB region to be registered. In our case this region will be only one row, as will be explained in Section 4.1.4.

Figure 3.8 Message flow between the AgentX master agent and AgentX subagent

b) SNMP request processing: These PDUs are related to the SNMP request processing. AgentXGetPDU, AgentXGetNextPDU, and AgentXSetPDU contain the SNMP requests from the manager (specifying the OID of the variable requested), and AgentXResponsePDU returns the variable's syntax and value.

c) Session closing: These PDUs unregister the MIB region specified and close the connection between the subagent and the master agent.

- **AgentXUnregisterPDU:** This PDU specifies to the master agent the region to be unregistered.
- **AgentXResponsePDU:** The first AgentXResponsePDU is used by the AgentX master agent to confirm that the MIB region specified in the AgentXUnregisterPDU has been unregistered. The master agent confirms the unregistration by returning the first and last OID of the region unregistered. The second AgentXResponsePDU returns the SessionID field set to 0 to specify that the connection has been closed.
- **AgentXClosePDU:** The subagent passes to the master agent all the information of the session it wants to close.

Figure 3.8 does not depict the message flow when the manager issues an SNMPSet request. Figure 3.9 depicts the message flow between an AgentX master agent and an AgentX subagent when a manager issues an SNMPSet request and this request is successfully accomplished.

Figure 3.9 SNMP Set request message flow

3.2.4 Security

AgentX is a protocol between two processing entities one of which, the master agent, is assumed to perform authentication of received SNMP requests and to control access to management information. The master agent performs these security operations independently of the other processing entity (the subagent).

AgentX does not contain a mechanism for authorizing/refusing session initiations. Any subagent can start a session with a particular master agent.

As mentioned earlier, AgentX was originally designed to have both the AgentX master agent and the subagents running on the same machine. In this case, where a local transport mechanism is used, connection authorization can be delegated to the operating system features. However, if a network transport is used, there is no inherent security.

During an AgentX session no SNMP security-related information (for example, community names) is passed from the master agent to the subagent. All of the access control process takes place at the master agent. A subagent can register any subtree (subject to AgentX elements of procedure). Currently there is no access control mechanism defined in AgentX.

In summary, AgentX does not provide any access control mechanism. A subagent can register any subtree. However, SNMP does provide access control. When requesting managing information from a subagent, the access control will be done at the master agent before forwarding the request to the subagent. AgentX provides a mechanism for transmitting SNMP context specifications in its PDUs. Nonetheless, it does not place any constraints on the content of that specification.

3.3 Implementations of AgentX

An AgentX Implementation Report Summary can be found in [33]. A comparison of some implementations is given in Table 2.3. Among the implementations presented I emphasize JAX [31] and J.AgentX [34] because they are the only two toolkits (TKs) whose language is Java.

Implementation	MultiNet	AgentX++	J.AgentX	SNMP Library	JAX	NET-SNMP
TK Language(s)	C	C++	Java	C, C++	Java	C
Registration operations	Yes	Yes	Yes	Yes	Yes	Yes
Registration priorities	No	Yes	Yes	Yes	No	No
Index Allocation	No	Yes	No	No	No	No
Index Deallocation	No	Yes	No	No	No	No
Multiple subagent connections	Yes	Yes	Yes	Yes	No	Yes
Multiple master agent connections	No	No	No	Yes	No	No
Detect subagent connection loss	No	Yes	No	Yes	No	Yes
Detect master agent connection loss	No	Yes	No	No	No	No
TCP/IP (port 705)	Yes	Yes	Yes	Yes	Yes	Yes
Other Transport Mapping	No	No	Yes	Yes	No	Yes
Implemented RFC2742	Yes	Yes	Yes	No	No	No
Master agent MIB implementation	Yes	Yes	Yes	NA	No	NA

Table 3.1 Implementations of AgentX [33]

3.4 Java AgentX (JAX)

As mentioned in Section 3.3, there are two implementations of AgentX in Java: JAX and J.AgentX. I discuss them here in more detail due to the growing number of services implemented in Java and the rising acceptance of Java management solutions in general.

The J.AgentX [34] package has been developed at the University of Coimbra, Portugal, while JAX [31] was developed at TU Braunschweig, Germany. While JAX contains a MIB compiler that allows for hiding MIB details, such as OIDs and underlying AgentX types from the subagent developer, the user of the J.AgentX API has to handle this information manually (e.g. the methods for the registration of AgentX subagent and for handling VarBind need to be developed manually). Furthermore, higher-level handling of table row creation and deletion and support for sending notifications are not yet present in J.AgentX.

J.AgentX supports only SNMPv1 and UDP (User Datagram Protocol) based AgentX communication. It consists of two major components: An SNMP eXtensible agent and an AgentX-API for the development of SNMP subagents.

JAX [31] is a Java toolkit that allows the rapid development of AgentX subagents based on a core AgentX class package and classes generated by a MIB compiler. It also allows applications and services to dynamically extend an SNMP agent by new MIB objects using the AgentX protocol. It was designed to be small and simple. The motivation was to keep the overhead of management instrumentation as small as possible [31]. This is

important for Java programs that are dynamically delegated to remote execution environments. JAX is derived from the AgentX specification and does not carry any SNMP overhead with it. The JAX software consists of:

- A Java class library that represents AgentX connections, sessions, registrations, PDUs, and generic classes for MIB structures like scalar groups, tables, and notifications.
- A MIB compiler that allows for the creation of stub classes for the JAX-compliant implementation of a given SMI (Structure of Management Information) v1/v2 MIB module.

Management information is viewed as a collection of managed objects. Collections of related objects are defined in MIB modules. SMI MIB specifications are used to describe the structure and semantics of SNMP MIB objects. The JAX MIB compiler creates source code files of Java classes for every table and scalar group of a given MIB module. This compiler is built on top of the freely available SMI library libsmi [35]. The libsmi software uses the SMI library as a front-end to parse MIB modules into internal data structures. Afterwards, it calls a specific built-in output driver to dump MIB contents in one of several formats.

Using a MIB module as the input for the MIB compiler, one can create the stub classes that will be called when the MIB region requested by a manager is handled by an AgentX subagent, as mentioned earlier in Section 3.2.1.

JAX does not implement all of the design features of AgentX. It does not implement index allocation and capabilities registration. It does not contain an AgentX master agent either. It has been developed and tested against the AgentX master agent of the NET-SNMP project [32]. When looking at Table 2.5 one could think that other implementations have features that JAX does not have, such as “multiple subagent connections” or “detect subagent connection loss”. However, JAX also “borrows” these two features from NET-SNMP when using the NET-SNMP master agent. The other features, such as index allocation or registration priorities, are present neither in JAX, nor in NET-SNMP, but they are not necessary for the implementation given in this thesis.

Despite JAX’s downsides, I chose this toolkit to develop AgentX subagents, as will be described in Chapter 4, because it was designed to create small and simple subagents. Given the memory constraints that PDAs have this characteristic is fundamental.

CHAPTER 4

The pdaMIB Application

Presently, as networked wireless appliances and devices are becoming commonplace, there is a need for providing a standard way to manage the networks these devices form. In order to do this, I developed an application: the pdaMIB. The design outlined in this chapter provides an extension of the AgentX protocol that allows an administrator to manage small devices (such as PDAs) using SNMP.

This application and the tools described in Chapter 5 are presented as the prototype solution for the management of Ubiquitous Networks. I assume that devices in a Ubiquitous Environment have similar specifications as PDAs. That is, that they have similar memory and storage capacity, power consumption, etc. Even though the applications developed here are specific for PDAs, they could be extrapolated to any similar device in a Ubiquitous network.

This chapter is organized as follows. Section 4.1 explains the architecture of the pdaMIB application, design and implementation. Section 4.2 presents the PDA MIB implemented by the pdaMIB application. Section 4.3 depicts a possible execution scenario of the pdaMIB application. Section 4.4 presents the integration of JAX in small devices. The development and implementation environment of the application are described in Section 4.5. Section 4.6 addresses some of the limitations of the pdaMIB application. Finally, Section 4.7 provides a summary of this chapter.

4.1 PdaMIB: Application Analysis and Design

4.1.1 Application Architecture

The three applications developed in this thesis are: The ARAX-SNMP agent, the RAgentX tools and the AgentX subagent.

The ARAX-SNMP agent is a modified version of NET-SNMP agent. It is an AgentX and RAgentX enabled SNMP agent. The NET-SNMP version used in this thesis (version 5.0.6) already supports AgentX, and has been modified to support RAgentX. The RAgentX protocol and RAgentX tools will be discussed in Chapter 5.

The ARAX-SNMP agent receives SNMP messages from a remote manager at the UDP port 161. The AgentX master agent and RAgentX agent accept TCP connection requests for the well-known port 705. Subagents connect to the master agent using this port number [29]. This is depicted in Figure 4.1.

The subject of this chapter is the pdaMIB application which allows the manager to manage many devices in a UN, like one would manage any other NE with SNMP. This application is an AgentX subagent running on a small device. It implements a management MIB whereby it can provide requested information to a manager, and it can accept management requests to modify certain value in the MIB for configuration or operation management purposes.

Figure 4.1: ARAX-SNMP architecture

For the implementation of the AgentX subagent I have used JAX [31]. As explained in Chapter 3, JAX allows applications and services to dynamically extend an SNMP agent with new MIB objects using the AgentX protocol.

The ARAX-SNMP master agent provides multi-threading support for handling concurrent incoming messages asynchronously. It also allows third parties to develop MIB-extending subagents that can be started and stopped independently of the master agent or other subagents. Henceforth I will refer to the ARAX-SNMP master agent as simply the master agent.

The master agent allows different subagents to register the same MIB table with the result that different rows of the same tabular object can reside in different subagents. The MIB region that each subagent registers and implements is a row of the PdasTable. This table is described in Section 4.2, and in Section 4.1.4 is explained how the master agent allows several subagents to register different rows of the PdasTable. Figure 4.2 shows the pdaMIB application/ARAX-SNMP agent interaction model.

Figure 4.2 pdaMIB/ARAX-SNMP Agent Model

In the following subsection I explain in more detail the main classes of the pdaMIB application. In subsection 4.3 three execution scenarios are given.

4.1.2 Main Classes

The AgentX subagent resides in the small device and is written in Java. As discussed earlier in Section 4.1.1, it allows the dynamic extension of the ARAX-SNMP agent MIB.

The main classes of the AgentX subagent, shown in Figure 4.3, are:

- a) **PdaMIBDemo**: This class implements a thread that initializes the application and allows the user to exit at any time. First of all, it calculates the number that is going to uniquely identify the PDA. It then opens a connection (**AgentXConnection**) with the ARAX-SNMP agent and a session (**AgentXSession**). After, it registers the subagent MIB as a row of the table **pdasTable** (using the unique identifier) (a more detailed explanation about how this unique number is calculated and the **pdasTable** is given in Section 4.1.4). When the user decides to exit the program, **PdaMIBDemo** unregisters the subagent MIB, closes the session, and closes the connection.
- b) **AgentXConnection**: It contains all the information referring to the connection between the subagent and the ARAX-SNMP agent. To open the session it sends an “**AgentXOpenPDU**” message to the ARAX-SNMP agent and starts the **Thread2 (AgentXReader)**. It is its’ responsibility to send all the PDUs directed to

the ARAX-SNMP agent through the open connection. This subagent will be connected to the local AgentX master through a TCP connection.

c) AgentXReader: This class implements a second thread. Its' role is to listen on the socket waiting for incoming PDUs sent by the ARAX-SNMP agent, to check which type of PDU has been received, and to call the function that handles this PDU. The main loop for the session process is:

- 1) waits for PDUs;
- 2) parses the PDUs received; and
- 3) generates reply messages.

d) AgentXRegistration: This class contains the information about the MIB registered by the AgentX subagent residing in the PDA.

e) AgentXPDU: This class represents a generic AgentX PDU. It is not used directly. Its methods are called from its subclasses. JAX implements all the AgentX PDU types.

The last four classes described form part of the JAX [31] class library. Here below are the classes that, with the PdaMIBDemo class, have been defined for the implementation of the pdaMIB application.

- f) PdasTable and PdasEntry: The PdasTable represents the PDA MIB table, and the PdasEntry is an entry in the PDA MIB table. They are both generated automatically by the MIB compiler.
- g) PdasEntryImpl: This class inherits from PdasEntry. It is the class responsible for providing access to management information. For this, it accesses to low-level information of the PDA. It implements the MIB region, the objects of the row, for which the subagent is willing to take responsibility.

The relationship between some of the classes of the pdaMIB application in the AgentX subagent side is shown using the class diagram in Figure 4.3. A more detailed class diagram with all the classes provided by JAX can be found in [31].

Figure 4.3 AgentX subagent side Class Diagram

4.1.3 Multi-threading

It can be seen from the class diagram in Figure 4.3 that there are two classes that implement threads. The following is a more specific list of the steps these two threads implemented.

Thread 1 (PdaMIBDemo)

1. Obtains the information that is going to be used to uniquely identify the PDA;
2. Opens a connection (AgentXConnection) with the ARAX-SNMP agent, and AgentXConnection starts the Thread2 (AgentXReader);
3. Opens a given AgentX session within this connection (AgentXSession);
4. Registers the subagent MIB using the unique identifier calculated previously;
5. Waits until the user decides to exit the program;
6. Unregisters the subagent MIB;
7. Closes the session; and
8. Closes the connection.

Thread 2 (AgentXReader)

While the connection is open the Thread2:

1. Listens to the socket;
2. Reads from the socket and parses the PDUs received;
3. If the packet received is an AgentXResponsePDU, it then
 calls the operation notify() of the field 'connection' of Thread1;
else
 calls the function that handles the PDU

As can be seen from the above description, the role of Thread1 is that of “interaction” with the user, (e.g. to start the application and to wait until the user decides to finish it).

On the other hand, Thread2 is in charge of the communication with the SNMP agent. The two threads run independently, but they need to be synchronized at two moments: a) when the user establishes the connection, Thread1 initiates it, but it is Thread2 which receives the response indicating if the attempt was successful or not; and b) when the user exits the application, Thread1 notifies Thread2 that the connection is closed. The communication between the two threads is as follows:

- Thread1 (PdaMIBDemo) waits until Thread2 (AgentXReader) calls “notify()” when opening a session, registering, and unregistering the MIB;
- Thread2 uses the boolean variable “AgentXConnection.open” to know if the connection is still open or not. When the user starts the application, this variable is given the value ‘true’, and when the user exits the application, Thread1 changes the value of the variable to ‘false’.

The following figures (Figure 4.4 to 4.7) show part of the code of the application described above.

From PdaMIBDemo (Thread1)

```
public PdaMIBDemo(String argv[])
{
    ...
    int i = OS.DllGetSyncInfo(...,nameBufP,...); //Gets the unique identifier
    String cadena = nameBufP.getString();
    row = cadena.hashCode(); //gets the row, using the PDA identifier

    long[] value = { 1, 3, 6, 1, 4, 1, 2000, 1, 1, 1, row};

    try {
        connection = new AgentXConnection(host, port);
        session = new AgentXSession();
        connection.openSession(session);

        registration = new AgentXRegistration(new AgentXOID(value));
        short range = 11;
        registration.setRangeSubID(range);
        registration.setUpperBound(6);
        session.register(registration);

    } catch (Exception e) {}

    pt = new PdasTable();
    session.addGroup(pt);
    pe1 = new PdasEntryImpl(row);
    pt.addEntry(pe1);
}
```

Figure 4.4 Code Segment from the PdaMIBDemo class (I)

```
public void run() {
    ...
    while(true) {
        Thread.yield();
        ...
        OS.EvtGetEvent(event, 1); // Get the events
        if (OS.SysHandleEvent(event) == ctrue)
            break; // checks if the user wants to exit
    } //while

    try {
        session.unregister(registration);
        session.close(AgentXSession.REASON_SHUTDOWN);
        connection.close();
        /* changes the value of the variable 'open' to FALSE
        to indicate to the AgentXReader that the connection is closed */
    } catch (Exception e) {}
}
```

Figure 4.5 Code Segment from the PdaMIBDemo class (II)

From AgentXConnection

```
private void open(InetAddress iaddr, int port, AgentXSignal signaller)
throws IOException
{
    socket = new Socket(iaddr, port);
    reader = new AgentXReader(this, signaller);

    open = true; // Marks the connection open
    reader.start(); //starts the AgentXReader
}
```

Figure 4.6 Code Segment from the AgentXConnection class

From AgentXReader (Thread2)

```
public void run(){
...
try {
    Thread.yield();
    ...
    while (connection.open){ //checks if the connection is still open
        readCount = input.read(recvPacket); //reads from the socket

        try {
            ...
            pdu = parsePDU(recvPacket);
        } catch (ParseException e) {}

        if (pdu instanceof AgentXResponsePDU) {
            session = connection.getSession(0);
            if (session != null) //seems a response to an Open PDU
                session.setSessionID(pdu.getSessionID());

            synchronized (connection) { // let the main thread go on
                connection.setReceivedPDU(pdu);
                connection.notify();
            }
        } else {

            session = connection.getSession(pdu.getSessionID());
            if (pdu instanceof AgentXGetPDU) {
                session.handleGetPDU((AgentXGetPDU)pdu);

            } else if (pdu instanceof AgentXGetNextPDU) {
                session.handleGetNextPDU((AgentXGetNextPDU)pdu);

            } else {
                AgentXDebug.debugMessage("reader cannot handle PDU");
            }
        }
    }
} catch (IOException e) {
    AgentXDebug.debugMessage("reader died: " + e);
}
...
}
```

Figure 4.7 Code Segment from the AgentXReader class

4.1.4 Index Allocation and PDA's Unique Identifier

This section describes how AgentX subagents use the AgentXRegisterPDU to achieve desired configurations.

In UNs there are devices, (such as PDAs, handhelds, etc.) with similar characteristics. If the AgentX subagents of these devices want to register with an ARAX-SNMP agent, the information for which they will be willing to take responsibility will then be very similar as well. When implementing the MIB for a UN, devices with similar characteristics will be grouped together and will make reference to a common MIB table. This is what is

referred to as “sharing of MIB conceptual tables.” The MIB table will contain the information that can be requested from a device, and each device will be registered as a row of that common table. The columns of the MIB table will represent the variables that can be requested from a device, and each one of the rows of the table will represent a different device. A row and a column of the MIB table represent a specific MIB variable of a specific device.

To register only a row in a MIB table I will use the ability of AgentX to register sub-id ranges. This will allow me to register rows of a table separately (by separate subagents).

The ability of registering rows independently facilitates enormously the implementation of MIBs for UNs. It is not necessary to have a different MIB for each device, but simply a MIB table that represents a group of them.

In my implementation, each AgentX subagent will register only one row. When a subagent sends an AgentXRegisterPDU to register with the AgentX master agent, the OID of this PDU will contain the number of the row for which the subagent will take charge of.

Index allocation is a service provided by AgentX master agents, as explained in Section 3.2.1. It provides generic support for sharing MIB conceptual tables among subagents who are assumed to have no knowledge of each other. This is a very necessary service in our case due to the fact that the devices need to determine the row(s) for which they will

take responsibility without knowing the rows other devices are registering. The master agent maintains a database of index objects (OIDs), and, for each index, the value that have been allocated for it.

As mentioned in Section 3.2.1, to avoid conflicts when several subagents want to register rows in the same MIB table, AgentX provides two solutions:

- If the index to be allocated has an intrinsic meaning, the master agent allocates the value provided by the subagent.
- If the index to be allocated does not have an intrinsic meaning, the master agent solves the conflict by assigning to the index a value that has not been allocated yet.

The ARAX-SNMP agent does not support index allocation, because it is based on NET-SNMP, and NET-SNMP does not support index allocation. Furthermore, index allocation is not implemented in JAX version 0.0.15, the version that I used and the most recent release.

In the case I am considering in this thesis, with a MIB table that represents different devices and with a master agent that does not support index allocation, I only have the option to find a value that will uniquely identify each device and can be used as an index with intrinsic meaning.

In the case of PDAs, the “serial number of the ROM” can be used as a unique identifier, or assuming that the users have different names, one can take as the PDA’s identifier the name that the PDA uses to synchronize. To simplify the implementation I have chosen the second option. To uniquely identify the PDAs I will use the name that the PDAs use to synchronize.

The two steps that are followed to calculate the unique identifier for a PDA are (the following lines of code can be also seen in Figure 4.4):

1. Get synchronization information (the name that the PDA uses to synchronize);

```
int i = OS.DllGetSyncInfo( ... ,nameBufP, ... );  
String cadena = nameBufP.getString();
```

2. Calculate the hash code of the string that contains the synchronization information. The calculated hash code will determine the row that the subagent will use to register⁴.

```
row = cadena.hashCode();
```

Once the device identifier has been calculated, the subagent sends the request to register the row using the device identifier. Consider now the MIB table pictured in Figure 4.8.

⁴ Two subagents cannot register the same row. If two strings produce the same hash code one of the subagents will not be able to register, and will have to recalculate the hash code.

Figure 4.8 Tree representation of a MIB table

To implement row 234 of the 'pdasTable', the subagent will attempt to register by specifying a MIB region in the AgentXRegisterPDU. The MIB region will be:

1.3.6.1.4.1.2000.1.1.1.[1-6].234

Registering this region is equivalent to registering the union of the following subtrees.

1.3.6.1.4.1.2000.1.1.1.1.234
1.3.6.1.4.1.2000.1.1.1.2.234
1.3.6.1.4.1.2000.1.1.1.3.234
 ⋮
1.3.6.1.4.1.2000.1.1.1.6.234

To register a MIB region with a single PDU, the AgentXRegisterPDU should contain these three variables: OID, RangeSubID and UpperBound. The OID specifies the first OID of the region, the RangeSubID specifies the sub-identifier of the subtree where the

subrange is located, and UpperBound indicates the number of columns in the table that will be registered. RangeSubID and UpperBound provide a general shortcut mechanism for specifying a MIB region. The UpperBound of a sub-identifier's range is present only if RangeSubID is not 0.

In the above example, the OID that an AgentX subagent sends as part of the AgentXRegisterPDU to register the row 234 with the AgentX master agent is:

OID: 1.3.6.1.4.1.2000.1.1.1.1.234[] RangeSubID:11 UpperBound:6

The OID of the subtree to register is 1.3.6.1.4.1.2000.1.1.1.1.234 because this is the first OID of the region, and the UpperBound is 6 (the 6 columns of the table that the subagent wants to register). The RangeSubID is 11 because the subrange, [1-6], is in the 11th sub-identifier of the subtree (1.3.6.1.4.1.2000.1.1.1.[1-6].234).

When the SNMP agent is implemented correctly, this information sent in the AgentXRegisterPDU will register row 234. From then on, until the subagent unregisters that row, the master agent will forward to the subagent any SNMP requests that refer to row 234.

4.2 PDA's MIB

As mentioned in 4.1.4, when implementing the MIB for a UN, devices with similar characteristics (such as PDAs, toasters, televisions, etc.) will be grouped together and will make reference to a common MIB.

I have defined a MIB that could be used in the management of PDAs. This MIB includes information about the OS version, number of databases, percentage of power remaining in the battery, number of cards, and battery type of the PDA. If I had defined a MIB for the management of televisions, for example, the information the MIB would include would be: the state of the TV, the number of channels available, etc.

The MIB I have developed for the management of PDAs is:

```
PDA-SMI DEFINITIONS ::= BEGIN

IMPORTS
    MODULE-IDENTITY, OBJECT-IDENTITY, enterprises
        FROM SNMPv2-SMI;

pdas MODULE-IDENTITY
    DESCRIPTION
        "The initial revision."
    ::= { enterprises 2000 }
END

PDA-MIB DEFINITIONS ::= BEGIN
IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE, Unsigned32,
    Integer32
        FROM SNMPv2-SMI
    DisplayString
        FROM SNMPv2-TC
    pdas
        FROM PDA-SMI;

pdasMIB MODULE-IDENTITY
    LAST-UPDATED "200303050000Z"
    ORGANIZATION "Carleton University"
```

CONTACT-INFO

"Catalina Diaz Puig
Carleton University
Ottawa, Ontario
Canada
Tel: + (613) 520-2600 ext. 3548
E-mail: catalina@sce.carleton.ca"

DESCRIPTION

"The initial revision of this module."
::= { pdas 1 }

pdasTable OBJECT-TYPE

SYNTAX SEQUENCE OF PdasEntry
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"A list of the PDA's features."
::= { pdasMIB 1 }

pdasEntry OBJECT-TYPE

SYNTAX PdasEntry
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"An entry containing a PDA's features."
INDEX { pdasIndex }
::= { pdasTable 1 }

PdasEntry ::=

SEQUENCE {
 pdasIndex Integer32,
 pdaOSVersion DisplayString,
 pdaNoDB INTEGER,
 pdaNoCards INTEGER,
 pdaBatteryLevel INTEGER,
 pdaBatteryType INTEGER
}

pdasIndex OBJECT-TYPE

SYNTAX Integer32 (1..4294967296)
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"A unique value identifying the pda entry."
::= { pdasEntry 1 }

pdaOSVersion OBJECT-TYPE

SYNTAX DisplayString (SIZE (0..255))
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"Version of the OS."
::= { pdasEntry 2 }

pdaNoDB OBJECT-TYPE


```

SYNTAX    INTEGER (1..255)
MAX-ACCESS read-only
STATUS    current
DESCRIPTION
    "Number of DB in the PDA."
 ::= { pdasEntry 3 }

pdaNoCards OBJECT-TYPE
SYNTAX    INTEGER (1..127)
MAX-ACCESS read-only
STATUS    current
DESCRIPTION
    "Number of Cards in the PDA."
 ::= { pdasEntry 4 }

pdaBatteryLevel OBJECT-TYPE
SYNTAX    INTEGER (1..255)
MAX-ACCESS read-only
STATUS    current
DESCRIPTION
    "A value that indicates the Level of the battery of the PDA.
     Percentage of power remaining in the battery."
 ::= { pdasEntry 5 }

pdaBatteryType OBJECT-TYPE
SYNTAX    INTEGER (1..255)
MAX-ACCESS read-only
STATUS    current
DESCRIPTION
    "Type of battery in the PDA."
 ::= { pdasEntry 6 }

END

```

A graphic representation of this MIB is pictured in Figure 4.8.

4.3 Execution Scenarios

NET-SNMP provides a number of tools for the administrator to manage the network. I have tried all of them with my application. The most significant ones are: `snmpget`, `snmpset`, `snmpbulkget`, `snmpbulkwalk`, `snmpdelta`, `snmptable`, and `snmpstat` (with the

options get, set, etc.). The following scenarios provide a comprehensive picture of how pdaMIB works.

Execution Scenario 1

In this scenario, there are three different PDAs intending to register one row of the pdasTable (see Figure 4.8). PDA 'A' will register row 65, PDA 'Z' will register row 90, and the third PDA, PDA 'T', will register row 84. For the relationship between the PDA's name and the row it implements refer to Section 4.1.4.

When the users of the PDAs start running the application, a connection with the machine where the AgentX master agent is running will be open. The flow of messages between the subagent and the master agent begins as depicted in Figure 3.8. The subagent sends the AgentXOpenPDU to open the connection with the master agent, and waits for a response from the master agent, AgentXResponsePDU, confirming that the connection has been established. The subagents proceed to register, sending an AgentXRegisterPDU, specifying the row for which they want to be responsible, in other words, the management information they implement and for which they can accept management requests from a manager. In this example, the AgentXRegisterPDUs will contain rows 65, 90 and 84, for PDA 'A', PDA 'Z' and PDA 'T' respectively.

Once connected and registered, the manager can issue any SNMP command mentioned above, until the user quits the application. When the user quits the application, the subagent deregisters the row and closes the connection (see last messages in Figure 3.8.)

In this execution scenario, the three phases of communication between the subagent and the master agent (session opening, SNMP request processing, and session closing) has been explained. These phases were explained in more detail in Section 3.2.3, and pictured in Figure 3.8.

In the following two execution scenarios, the two “administrative” phases (session opening and session closing) will not be explained again. I will consider that there are three PDAs that have already registered rows 65, 90 and 84 (as explained above), and I will concentrate in the second phase of communication between the subagent and the master agent: the SNMP request processing.

Execution Scenario 2

Let us now consider what the procedure would be if the manager issues the following command shown in Figure 4.9. (For this scenario I consider that a PDA has registered the row 65 of the pdasTable).

```
snmpget -m all -c public 134.117.60.65 pdas.pdasMIB.pdasTable.pdasEntry.pdaOSVersion.65
```

Figure 4.9 SNMPGet command

This command is a SNMP Get request. The manager is querying the value of a variable using its representative name. Any variable can be queried using either its representative name, or its numeric OID. In this case, the representative name of the variable is “pdas.pdasMIB.pdasTable.pdasEntry.pdaOSVersion.65”, and the numeric OID is

“1.3.6.1.4.1.2000.1.1.1.2.65” (see Section 4.2). Both represent the 65th row of the table pdaSTable, column pdaOSVersion (the OS version of the PDA that registered row 65).

The ‘-m all’ option of the snmpget command is used because the variable requested belongs to a MIB that has been added dynamically. The ‘-c’ option is used to specify the community name, that is a plain text password. In our case I use ‘public’ as the community name because it is the default community name. After the community name the host name or IP address where the SNMP agent is running should be specified. In this case, the ARAX-SNMP agent is running on “134.117.60.65”.

Upon receipt and initial validation of the snmpget request, a master agent dispatches an AgentXGetPDU to the subagent that implements that variable. (As mentioned in Section 4.1.4, the master agent maintains a database of index objects with the values that have been allocated for them). The master agent identifies the target MIB region for the variable binding in the snmpget request, and sends an AgentXGetPDU with this VarBind requested. In this example, the master agent will look for the subagent that registered row 65, and will send an AgentXGetPDU to PDA ‘A’ requesting the value of the pdaOSVersion MIB variable.

When a subagent receives an AgentXGetPDU, if the received PDU's header was successfully parsed, it performs the indicated management operations and returns an AgentXResponsePDU. If the variable requested matches the name of a variable instantiated by this subagent, the variable's syntax and value are encoded in the

AgentXResponsePDU. Otherwise, the VarBind is set to 'noSuchObject'. In this example, the subagent running on PDA 'A' will try to match the variable requested, pdaOSVersion, with the variables instantiated. The variable has been instantiated in PDA 'A', the variable's syntax 'STRING' and the variable's value 'v. 4.1' will be encoded in the AgentXResponsePDU, and the PDU will be sent to the master agent.

After receiving the AgentXResponsePDU from the subagent, the master agent encapsulates the PDU into an SNMP message, according to the version of the SNMP protocol in use, and transmits it to the originator of the SNMP management request. Then, the manager will be presented with the following response:

PDA-MIB::pdaOSVersion.65 = STRING: v. 4.1
--

Execution Scenario 3

Among the available SNMP tools, snmptable is a very representative one, because it shows a very important characteristic of the PDA's MIB: the fact that several devices implement different rows of the same MIB table. This command obtains a table that represents the variables' name and values of a MIB table.

If there are three PDAs that have registered rows 65, 90 and 84, as explained in execution scenario 1, let us now consider the procedure when the manager issues the command depicted in Figure 4.10.

```
snmptable -m all -c public 134.117.60.65 pdas.pdasMIB.pdasTable
```

Figure 4.10 SNMPTable command

With this command the manager is querying all the variables contained in “pdas.pdasMIB.pdasTable”, that is all the values of the table “pdasTable.” The options used with this command are the same as the ones used with snmpget explained above.

Eighteen variables are been requested with the above command: there are six columns (six variables) and 3 rows (since there are three PDAs, and each one of them registered a row). When processing the above request, each subagent will receive 6 AgentXGetPDUs, one per variable of the table of the row it implements. The rest of the procedure to request the MIB variables will be the same as in the “snmpget” example above. The result obtained and presented to the manager will be as shown in Figure 4.11.

pdaBatteryType	pdaBatteryLevel	pdaNoCards	pdaNoDB	pdaOSVersion	pdasIndex
2	80	1	117	v. 4.1	65
2	50	1	132	v. 4.1	90
5	100	1	83	v. 3.5.0	84

Figure 4.11 Results from a snmptable request

Figure 4.12 depicts the message flow between the manager, the AgentX master agent and the AgentX subagents residing in PDA ‘A’, ‘Z’ and ‘T’, when the manager issues the ‘snmptable’ command explained above and depicted in Figure 4.10. The message flow

between the master agent and PDAs 'Z' and 'T' is equivalent to the message flow between the PDA 'A' and the master agent. The "SNMP processing request" phase for PDA 'A' depicted in Figure 4.12 is the same as for the other two PDAs, although the variables requested are different. The "snmptable" command is issued only once, and the manager receives only one 'snmpresponse' with the responses from the three PDAs.

Figure 4.12 Message flow between a master agent and the subagents when an 'snmptable' command is issued

4.4 Modifying JAX for Small Devices

I chose Java as the programming language for the pdaMIB application and the RAgentX tools described in Chapter 5. Running on a PDA, the KVM will execute the application developed. As mentioned earlier in Section 2.3.1, the KVM's libraries are not 'complete' libraries, but rather a subsection of the ones in a regular JVM.

To develop my application I used VAME (VisualAge Micro Edition) introduced in Section 2.3.2, particularly the jclPalmXtr for PalmOS library provided by VAME.

As already mentioned, my application is based on JAX. The classes that the JAX package uses have been modified to adjust to what is provided by the jclPalmXtr library.

The changes that had to be done to adjust the JAX library to the classes and methods provided by the jclPalmXtr library are:

- I created a new class, `VectorBis`, extending the class `Vector` (`java.util.Vector`). `VectorBis` implements two methods that are not present in the version of `Vector` provided by the jclPalmXtr library:

<code>isEmpty()</code> <code>elements()</code>

Whenever the class `Vector` was used in the JAX library, I used `VectorBis` instead. This is the case for the following classes: `AgentXConnection`, `AgentXPDU`, `AgentXNotification`, `AgentXReader`, `AgentXScalars`, `AgentXSession`, and `AgentXTable`.

- The library jclPalmXtr does not provide a constructor with 0 arguments for the class String.

```
String s = new String();
```

Each time this constructor was used in the JAX library, it was substituted by:

```
char [] aux = {};  
String s = new String(aux,0,0);
```

For example, this is the case with the classes AgentXOID, AgentXRegistration, AgentXSession, and AgentXVarBind.

- The jclPalmXtr library does not provide a constructor with only one argument (an array of byte) for the class String.

```
byte[] value_bytes;  
s = new String(value_bytes);
```

Therefore, each time this constructor was used in the JAX library, it was substituted by:

```
s = new String(value_bytes,0,value_bytes.length);
```

This new constructor is used only in the class AgentXVarBind.

- The jclPalmXtr library does not include the class:

```
ArrayIndexOutOfBoundsException
```

Whenever this class was needed, it was substituted for:

IndexOutOfBoundsException

This did not present any problem because the `ArrayIndexOutOfBoundsException` class extends `IndexOutOfBoundsException` and only uses the inherited methods. I did this substitution for the following classes: `AgentXSession` and `AgentXTable`.

- The class `PdaMIBDemo` extends `Thread` and implements `OSConsts`. The `OSConsts` is a public Interface part of the `jclPalmXtr` library. The `PalmOS` API is written in C. This interface wraps all the C constants of the `PalmOS` API.
- The class `AgentXVarBind` makes use of the method "`toCharArray()`" of the class `String` that is not implemented in the `jclPalmXtr` library. To solve this problem, I defined a function that does exactly the same as the omitted method.

```
public char[] toCharArray(String value) {  
    char result[] = new char[value.length()];  
    value.getChars(0, value.length(), result, 0);  
    return result;  
}
```

Each time when in the `JAX` library this method was called for,

```
String value;  
char[] c = value.toCharArray();
```

it was substituted for:

```
char[] c = toCharArray(value);
```

- AgentXConnection and AgentXReader make use of the class ParseException, (`java.text.ParseException`) that is not implemented in the jclPalmXtr library. I created a new class `util.ParseException` that implements the methods of `java.text.ParseException` used in the JAX library.

4.5 Development and Implementation Environment

In this section, I discuss some of the features of the environment in which the pdaMIB application has been implemented and developed, and the reasons for using some of these features. Most of them have been chosen as the most suitable for the characteristics of the pdaMIB application.

The pdaMIB application has been developed to allow PDAs in a UN system to be managed. An example of a UN system could be a household environment. If the AgentX framework is applied to this household environment, there will be one AgentX master agent (the ARAX-SNMP agent) and multiple subagents running on PDAs.

The ARAX-SNMP agent is based on NET-SNMP [29], and written in C. To run the pdaMIB application no changes were needed in NET-SNMP since version 5.0.6 of NET-SNMP supports AgentX.

I have used the smidump tool, which is part of the libsmi software distribution [35], to compile MIBs to JAX [31] conformant Java classes that represent the MIB scalar groups, tables, and notifications. I used libsmi version 0.4.0. and JAX 0.0.15.

The communication between the PDA and the ARAX-SNMP agent is wireless communication based on IEEE 802.11b. As mentioned in Section 2.2.3, other technologies that could have been used are: IrDA, Bluetooth, etc. However, I considered that the IEEE 802.11b technology, that allows the user to access the SNMP agent from anywhere in a household, is the most suitable for the purposes of the pdaMIB application.

The application pdaMIB has been written in Java. There are several tools that can be used to develop applications in small devices using Java. These tools were presented in Section 2.3.2. For the development of the pdaMIB application, I have chosen VisualAge Micro Edition (VAME) [27] because VAME provides full access to the Palm OS API. VisualAge provides libraries that allow the access to low-level characteristics of the PDA. This feature is very important given that the application needs to return the value of the requested MIB variable, and that these variables are low-level information.

A very important challenge when programming for small devices is the footprint of the applications developed, and when written in Java, the footprint of the JVM. In my environment, the size of the KVM, CLDC and Profile (see Section 2.3.1) used is 223KB. The KVM used is J9 VM from IBM, version 1.4.

The OS of the PDA where the application runs is PalmOS. This OS has been chosen because it is the most common OS of the PDAs in the market. The application has been tested with emulators of a Palm m505 handheld (16MB RAM, 8MB ROM, Palm OS 4.1), and a Palm IIIc handheld (8MB RAM, 8MB ROM, Palm OS 3.5).

4.6 Limitations

The following are some of the limitations of the pdaMIB application. Even though Java claims to be "universally" portable, currently this is not completely true for small devices. Different companies provide different KVM based on different subsections of the libraries in a regular JVM. As a consequence, the paradigm "write once, run anywhere" is not true. In this thesis, what makes the pdaMIB application device dependent is the fact that it accesses information that is low level. The pdaMIB application can only be run on devices with J9 KVM from IBM.

For the development of the pdaMIB application a non device dependant application was also developed. This application runs on a regular JVM. However, this application does not implement any MIB. As mentioned earlier, what it makes the pdaMIB application device dependant is the implementation of a MIB.

I have implemented a resource-constrained compatible device-specific client that is based on the J2ME suite for Palm devices with Wireless LAN support. The fact that the application is based on IEEE 802.11b can be seen as a limitation because all current

PDA's come with infrared facilities, but not with wireless cards. However, I considered that, given the pace at which the technology is progressing, very soon most PDA's will come with wireless cards incorporated. At the same time, I considered that this application is more useful if the user can access the SNMP agent in a range of 100 meters.

4.7 Summary

At the beginning of this chapter, I have analyzed the characteristics of the pdaMIB application. I have described the design and development process of the application. The program architecture, main classes, operations sequence and algorithms are also described.

The pdaMIB application is a client server network application. The client, an AgentX subagent, is developed to run on PDA's using J9 VM (a KVM from IBM). The pdaMIB application allows the dynamic extension of the SNMP MIB. Some of the main classes of the application are: PdaMIBDemo, AgentXConnection, AgentXReader, and PdasEntryImpl. These classes enable the subagent to connect with an ARAX-SNMP agent, to register a MIB region, and to exchange messages with the master agent. The PdaEntryImpl class is the class that accesses low-level information. Using these classes a manager (through a master agent) can request any variable in the MIB region implemented by the subagent.

Due to the constraints of the palm computer and Java J9 VM, some particular issues are addressed. It was necessary to modify JAX to adjust to what is provided by the jclPalmXtr library, the VAME library for PalmOS devices.

Finally, I also discussed the application development environment and implementation issues, as well as some limitations of the application. The pdaMIB application has been developed using the jclPalmXtr library, the VAME library for PalmOS devices. This library makes the application, even though it has been written in Java, a non "universally" portable application. The pdaMIB application can only be run on PalmOS devices with J9 KVM from IBM.

CHAPTER 5

The RAgentX protocol and the RAgentX tools

In Chapter 4 it was described how a PDA with a pdaMIB application running on it, in the presence of an ARAX-SNMP agent can be managed using the AgentX protocol like any other NE of a network. In this chapter, I describe the tools (RAgentXGet and RAgentXSet) that make possible for a small device to manage the NEs of a network. I will also describe the protocol used for the communication between the small device and the SNMP agent when the device is acting as a manager, rather than as a subagent as described in Chapter 4. The reason for the 'R' in RAgentX (Reverse AgentX) is because of the renewal of rules

5.1 RAgentX Protocol

I have designed the RAgentX protocol to make the communication between small devices, like PDAs, and SNMP agents possible. This protocol allows a small device that has the RAgentX tools, described in Section 5.3, to act as a manager, and to request or set values of MIB variables that are accessible from the ARAX-SNMP agent. These variables are "accessible" from the ARAX-SNMP agent either because the agent itself implements them, or because they are implemented by AgentX subagents and the agent has access to them through the AgentX protocol.

The RAgentX protocol can be used instead of the SNMP protocol for the management of NEs in a network when this management is done from a small device. The RAgentX protocol is like SNMP in as much as it is a protocol used for the management of networks, but RAgentX is more resource-constrained compatible than SNMP (this is discussed in more detail in Chapter 6).

The RAgentX protocol is based on the AgentX protocol. RAgentX has the same PDUs as AgentX, and the elements of procedure of the RAgentX entities (RAgentX manager and RAgentX agent) are the same as the elements of procedure of the entities of AgentX, that is, RAgentX protocol entities execute the same actions when processing messages as AgentX entities do. A more detailed explanation is given in the following subsections.

5.1.1 RAgentX PDUs

The RAgentX protocol has 3 types of PDUs: RAgentXGetPDU, RAgentXSetPDU, and RAgentXResponsePDU. These are the basic PDUs for the management of any device. The RAgentX PDUs have exactly the same format as those of AgentX.

RAgentX PDUs consist of a common header, followed by a PDU-specific data of variable length. As with AgentX messages, RAgentX messages are not encoded using BER, but are transmitted as a continuous byte stream. The RAgentX PDU header is a fixed-format, 20-octet structure as depicted in Chapter 3, Figure 3.2.

The format of the PDUs varies depending on the PDU type. The layout of the RAgentXGetPDU, RAgentXResponsePDU, and RAgentXSetPDU are not explained in this section because they are the same as AgentXGetPDU, AgentXResponsePDU, and AgentXTestSetPDU which were explained in Section 3.2.2 and depicted in Figure 3.3, Figure 3.6, and Figure 3.7 respectively.

For the processing of AgentX Set request, the AgentX protocol has four different PDUs (as explained in Section 3.2.1): AgentXTestSetPDU, AgentXCommitSetPDU, AgentXUndoSetPDU, and AgentXCleanUpSetPDU. RAgentX does not make use of the three latter ones for reasons that will be explained in Section 5.1.3.

5.1.2 Message Flow

The interaction between the RAgentX manager (using the RAgentX tools explained in Section 5.3), and the RAgentX agent (an ARAX-SNMP agent) is depicted in Figure 5.1.

For each request issued by the RAgentX manager, the RAgentX agent responds with an RAgentXResponsePDU.

An RAgentXGetPDU is used by the RAgentX tools when the manager wants to issue a Get Request. To do this, the RAgentX manager opens a TCP connection with the RAgentX agent, and sends an RAgentXGetPDU. To this request, the RAgentX agent answers with an RAgentXResponsePDU that contains a VarBind list with the numeric OIDs of the requested variables, and their values. Once the response has been received,

the connection is closed. The manager can issue many commands over the same TCP connection. The connection is open only once, and it is closed after the manager has finished doing the management activity.

Figure 5.1 RAgentX protocol. Messages flow

An RAgentXSetPDU is used by the RAgentX tools when the manager wants to issue a Set Request. The RAgentXSetPDU contains the numeric OIDs of the variables to be set, as well as the new values to be assigned. To this request, the RAgentX agent answers with an RAgentXResponsePDU that contains only the RAgentX header.

5.1.3 Processing RAgentX messages

For the processing of RAgentXGetPDUs and RAgentXResponsePDUs, the steps followed are the same as the ones in AgentX. This is explained in more detail in Section 5.2.5. In this section I explain how SNMP and AgentX handle Set requests as an introduction of how RAgentX handles them.

The design of SNMP and AgentX calls for all variables in a SET request to be done "as if simultaneously", i.e. they should all succeed or all fail. However, in practice, the variables are handled in succession. Thus, if one fails, it is necessary to "undo" any changes made to the other variables in the request. AgentX employs test-commit-undo-cleanup phases to handle this "as if simultaneous" semantic, while some implementations of SNMP (like NET-SNMP) employ reserve1-reserve2-action-commit-free-undo phases. These two sequences of phases are depicted in Figure 5.2.

Figure 5.2 AgentX and SNMP Set phases

In SNMP the Set phases are not part of the specification of the protocol⁵. In the AgentX protocol these phases are mandatory. While in a traditional SNMP agent the MIB variables are locally implemented in the agent, in AgentX the MIB variables are not implemented in the agent, but in the subagents. When the subagents are not running on the same machine as the master agent and the communication between them is done over the network, there are more chances that something goes wrong in the processing of a Set request. Hence the need for the Set phases to be mandatory. These phases ensure that the variables are changed correctly.

The “Reserve” (from SNMP) and “TestSet” (from AgentX) phases are used to check the syntax of all the variables provided, (that the values being set are sensible and consistent), and to allocate any resources required for performing the SET. But in the implementation of the NET-SNMP agent this checking is actually done in two passes: Reserve 1, and Reserve 2, to allow for dependencies between variables. Besides the names, the only difference between the phases of AgentX and SNMP is that the first phase in AgentX (TestSet) is divided into two sub-phases (reserve1 and reserve2) in SNMP.

While in SNMP these phases are just “internal” steps in the agent needed for setting the variables, for the AgentX protocol the phases are not just “internal” steps, but there are PDUs that are sent within the different phases. The four Set PDUs mentioned in Section 5.1.1 are used to collectively perform the indicated management operation. An AgentXResponsePDU is sent in reply to each of the PDUs (except AgentXCleanupSetPDU), to inform the master agent of the state of the operation.

⁵ For this reason, it is not mandatory to implement them.

RAgentX does not make use of AgentXCommitSetPDU, AgentXUndoSetPDU, and AgentXCleanupSetPDU because, like in SNMP, the RAgentX manager only sends one Set PDU for each Set request, and the agent answers back with only one Response PDU. Similar to the SNMP Set request, the ARAX-SNMP agent goes through the phases depicted in Figure 5.2 to correctly process the RAgentX Set Request.

5.2 Integrating RAgentX in the NET-SNMP Agent: the ARAX-SNMP Agent

The AgentX and RAgentX enabled SNMP agent developed in this thesis is called ARAX-SNMP agent. The implementation of SNMP on which ARAX-SNMP agent is based is NET-SNMP version 5.0.6.

The NET-SNMP agent is AgentX enabled. In fact, it provides not only a built-in AgentX master agent, but also an AgentX subagent. In other words, the NET-SNMP agent can take two roles: if acting as an AgentX master agent, subagents can register their MIB with it, and if acting as an AgentX subagent, it can register its MIB with another AgentX master agent. These two acting roles are depicted in Figure 5.3.

The approach I have taken to modify the NET-SNMP agent to make it RAgentX enabled is the following. The RAgentX agent Interface in the ARAX-SNMP agent could be seen as the equivalent to the AgentX subagent Interface of the NET-SNMP agent depicted in Figure 5.3.

Figure 5.3 NET-SNMP Functional Architecture

On the one hand, the RAgentX agent handles requests from an RAgentX manager. On the other hand, this is what the subagent embedded in the NET-SNMP agent does as well. The AgentX subagent receives AgentX request PDUs (Get, GetNext or Set), processes them, and returns the results. The RAgentX agent gets RAgentX requests, that are equivalent in format and content to the AgentX requests, and then responds to the manager by sending RAgentXResponsePDUs that are also equivalent to the AgentX responses. This means that the NET-SNMP agent has already built in all the necessary routines to handle the RAgentX requests.

This parallelism can also be seen if one compares Figure 3.8, that depicts the message flow between an AgentX master agent and an AgentX subagent, and Figure 5.1 that depicts the message flow between the RAgentX manager and the RAgentX agent. The only difference is that RAgentX skips all the packets for opening and closing sessions of AgentX (depicted in Figure 3.8). A manager does not have to register with the agent, but a subagent does have to.

Taking advantage of this similarity, I was able to use the AgentX API implemented in the NET-SNMP agent to support the RAgentX protocol. In the following subsections I explain in more detail how this API was reused.

5.2.1 Modifications to the NET-SNMP Code

Several source code files of the NET-SNMP library were modified to make the SNMP agent provided by the NET-SNMP [32] package RAgentX enabled. The files modified are:

- `snmpd.c`: This C implementation file contains the main function.
- `ds_agent.h`: In this C header file the agent's default store registrations are defined.
- `master_admin.c`: This C implementation file contains the functions for the AgentX Administrative request handling.
- `subagent.c`: The routines for the AgentX subagent are implemented in this C code implementation file.

The main modifications I made to the code of the NET-SNMP libraries are in the “master_admin.c” file. The code of the four files modified is presented in Appendix A.

5.2.2 SNMP Agent Options

When the SNMP agent is started, several options can be specified (if verbose output is desired, if the agent is an AgentX master agent or a AgentX subagent, if an UNIX port has been specified, etc). More details on the SNMP agent daemon are given in Sections 5.2.4.

For this thesis, the relevant options are:

- The “-x ADDRESS” option specifies that the SNMP agent should listen for AgentX connections on the specified address. The SNMP agent is acting as an AgentX master agent.
- The “-X” option specifies that the SNMP agent is acting as an AgentX subagent.

To run the SNMP agent acting as an AgentX master agent the daemon should be started specifying the option “-x ADDRESS” as depicted in Figure 5.4. In this example, the AgentX master agent listens for incoming AgentX packets on the TCP port 705.

```
snmpd -x 705
```

Figure 5.4 SNMP agent daemon started as an AgentX master agent

To run the SNMP agent acting as an AgentX subagent the daemon should be started specifying the options “-x ADDRESS” and “-X” as depicted in Figure 5.5. As in the example above, in this example the AgentX subagent listens for incoming AgentX packets on the TCP port 705.

```
snmpd -x 705 -X
```

Figure 5.5 SNMP agent daemon started as an AgentX subagent

5.2.3 Main Storages Registration

The C header file “ds_agent.h” contains the definitions of the agent’s default store registrations, as mentioned earlier. The options chosen when the SNMP agent is started need to be stored. The variables defined in the “ds_agent.h” file are used to specify where these options are stored.

Let us look at an example. If the “-x ADDRESS” option is used, the “ADDRESS” should be stored so that the agent knows on which address it should listen. The “ds_agent.h” file contains the definition of “NETSNMP_DS_AGENT_X_SOCKET” that specifies where this “ADDRESS” is stored.

I have added to the “ds_agent.h” file the definition depicted in Figure 5.6. This variable is used to specify where is stored the information that says if the SNMP agent is acting as an RAgentX agent. In Section 5.2.4 I explain how this variable is used.

```
#define NetSNMP_DS_AGENT_RAGENTX-MASTER 15 /*1 if the RAgentX is being used*/
```

Figure 5.6 Definition of an agent's default store registration

5.2.4 SNMP Agent Daemon

In “snmpd.c” the main function of the SNMP agent application is implemented. The SNMP daemon or “snmpd” is the SNMP agent which binds to a port and awaits for requests. Upon receiving a request, it processes the request(s), collects the requested information and/or performs the requested operation(s) and returns the information to the sender. By default, snmpd listens for incoming SNMP requests only on UDP port 161. However, it is possible to modify this behaviour. If the “-x ADDRESS” option is specified, the SNMP agent also awaits for AgentX requests listening to the address specified in “ADDRESS”.

The basic steps followed by the SNMP agent are:

- Initializes the application
- Goes through the options specified storing the necessary information
- Waits in an infinite loop listening to the ports⁶ until the application is finished

When a request is received, the daemon invokes the established message handlers for incoming messages and awaits for more requests.

⁶ As mentioned earlier, by default the port the agent listens to is the UDP port 161. However, it would listen to more ports if this is specified with the options used when started.

I have added some code to the “snmpd.c” file so that the SNMP agent provided by NET-SNMP can communicate with an RAgentX manager. Figure 4.1 shows this SNMP agent/RAgentX manager architecture. As mentioned earlier, the ARAX-SNMP agent is the NET-SNMP SNMP agent AgentX and RAgentX enabled.

From here on I refer to the interface of the ARAX-SNMP agent that deals with the RAgentX managers as RAgentX agent. As explained above, the RAgentX agent interface can be seen as equivalent to the AgentX subagent interface. This interface is explained in Section 5.2.5.

The modifications I have made to the “snmpd.c” file to make the SNMP agent provided by NET-SNMP RAgentX enabled are:

- As mentioned earlier, when the SNMP agent is started several options can be specified. The array “options” in the “snmpd.c” file contains the letters that represent the different options with which the daemon can be invoked. I have added an option (R) to the list of possible options to specify that the SNMP agent is acting as an RAgentX agent. This option is added by including the letter “R” in the array “options”, as shown in Figure 5.7.

```
char options[128] = "aAc:CdD::fhHI:l:LP:qrRsvV-: "; /*Added -R option*/
```

Figure 5.7 Array with snmpd’s options

- I have also added code to the main function of the “snmpd.c” file so that when the option “-R” is used the variable that specifies if RAgentX is used is set to 1. This is depicted in Figure 5.8. The definition of the variable that specifies if RAgentX is used variable was depicted in Figure 5.6, and explained in Section 5.2.3.

```
case 'R': /*Added to support RAgentX */  
    netsnmp_ds_set_boolean(NETSNMP_DS_APPLICATION_ID,  
                           NETSNMP_DS_AGENT_RAGENTX_MASTER, 1);  
    break;
```

Figure 5.8 The variable that specifies if RAgentX is used is set to 1

To run the SNMP agent as an RAgentX agent the option “-x ADDRESS” and “-R” should be used as depicted in Figure 5.9. In this example, the RAgentX agent listens for incoming RAgentX packets on the TCP port 705.

```
snmpd -x 705 -R
```

Figure 5.9 SNMP daemon started as an RAgentX agent

When the daemon is started as depicted in Figure 5.9, the SNMP agent can interact not only with an RAgentX manager, but also with an AgentX subagent because it is acting not only as an RAgentX agent, but also as an AgentX master agent. This is depicted in Figure 4.1. More details about this issue are given in Section 5.2.5.

5.2.5 RAgentX Agent Interface

As mentioned in Section 5.2.1, the “master_admin.c” C implementation file contains the implementation of the functions for the AgentX Administrative request handling. One of these functions is “handle_master_agentx_packet” that is invoked when AgentX packets arrive to the AgentX port (by default this port is the TCP port 705).

The “handle_master_agentx_packet” function, that is part of the AgentX master agent API, has three main steps:

1. checks which type of AgentX PDU has been received (this is done using a switch statement)
2. invokes the routine that handles the type of PDU received
3. sends the response generated by the routines to the sender

When the PDU received is an AgentXGetPDU, an AgentXGetNextPDU, an AgentXTestSetPDU, an AgentXCommitSetPDU, an AgentXUndoSetPDU or an AgentXCleanUpSetPDU in the switch statement of the “handle_master_agentx_packet” function, no routine to handle these PDUs is invoked. This is due to the fact that an AgentX master agent never receives any of these types of PDUs. The switch statement of the “handle_master_agentx_packet” function is depicted in Figure 5.10.

When an RAgentX PDU arrives to the RAgentX port, the function “handle_master_agentx_packet” is also invoked. In order to handle RAgentX PDUs I

have added to the switch statement depicted in Figure 5.10 routine calls that handle these PDUs. This is depicted in Figure 5.11.

```
int handle_master_agentx_packet(...)
{
    .....
    switch (pdu->command) {
    case AGENTX_MSG_OPEN:
        asp->pdu->sessid = open_agentx_session(session, pdu);
        if (asp->pdu->sessid == -1) asp->status = session->s_snmp_errno;
        break;

    case AGENTX_MSG_CLOSE:
        asp->status = close_agentx_session(session, pdu->sessid);
        break;

    .....

    case AGENTX_MSG_GET:
        break;

    case AGENTX_MSG_GETNEXT:
        break;

    case AGENTX_MSG_TESTSET:
        break;

    case AGENTX_MSG_COMMITSET:
        break;

    case AGENTX_MSG_UNDOSET:
        break;

    case AGENTX_MSG_CLEANUPSET:
        break;

    .....

    default:
        asp->status = AGENTX_ERR_PARSE_FAILED;
        break;
    }
    .....
    return 1;
}
```

Figure 5.10 The original “handle_master_agentx_packet” function

The name of the routine invoked to handle the RAgentX PDUs and shown in Figure 5.11 is “handle_agentx_packet”. This routine is part of the API of the AgentX subagent. It has the same role as the “handle_master_agentx_packet” function but for the AgentX subagent. It also has the same three main steps: a) checks which type of AgentX PDU has

been received (this is done using a switch statement), b) invokes the routine that handles the type of PDU receive, and c) sends the response to the sender.

```
int handle_master_agentx_packet( .... )
{
    .....

    switch (pdu->command) {
        .....

    case AGENTX_MSG_GET:
        if (netsnmp_ds_get_boolean(NETSNMP_DS_APPLICATION_ID, NETSNMP_DS_AGENT_RAGENTX_MASTER) == 1){
            if (!handle_agentx_packet(operation, session, reqid, pdu, magic))
                printf("Problem with the 'handle_agentx_packet'\n");
            return 1;
        }

        .....

    case AGENTX_MSG_TESTSET:
        if (netsnmp_ds_get_boolean(NETSNMP_DS_APPLICATION_ID, NETSNMP_DS_AGENT_RAGENTX_MASTER) == 1){
            if (!handle_agentx_packet(operation, session, reqid, pdu, magic))
                printf("Problem with the 'handle_agentx_packet'\n");
            return 1;
        }

        .....

    default:
        asp->status = AGENTX_ERR_PARSE_FAILED;
        break;
    }

    .....

    return 1;
}
```

Figure 5.11 The modified “handle_master_agentx_packet” function

As mentioned above, I have reused the API of the AgentX subagent because of the parallelism between AgentX subagents and the RAgentX agents.

Before invoking the routine that handles the RAgentX packet, the function “handle_master_agentx_packet” checks if the SNMP agent is acting as an RAgentX agent. This is done by checking the value of the variable that specifies if RAgentX is

used. Figure 5.8 shows how the “NETSNMP_DS_AGENT_RAGENTX-MASTER” variable is set to true. As mentioned earlier this variable is set to true when the “-R” option is specified.

The RAgentX manager only issues Get and Set requests. In order to do this, the RAgentX manager sends to the RAgentX agent RAgentXGetPDUs and RAgentXSetPDUs. That is, the RAgentX agent only receives these two types of PDUs. For this reason, I only had to add routine calls when in the switch statement the type of the AgentX PDU was AgentXGetPDU or AgentXTestSetPDU, as it is shown in Figure 5.11.

The “handle_agentx_packet” function generates the response to be sent to the RAgentX manager and sends it out to the network.

In summary, the “handle_master_agentx_packet” function a) receives both AgentX and RAgentX PDUs, b) checks which type of PDU has been received, and c) calls the routine that handles the type of PDU received. Some code has been added to the “handle_master_agentx_packet” function to invoke the routine that handles the RAgentX PDUs.

The “handle_master_agentx_packet” function is part of the AgentX master agent interface and the “handle_agentx_packet” function is part of the AgentX subagent interface.

The type of the RAgentX PDU received is checked twice. First it is checked in the “handle_master_agentx_packet” function, and after it is checked in the “handle_agentx_packet” function. This might appear redundant. Furthermore, it might seem redundant the fact that a function of the AgentX master agent interface is called first and after a function of the AgentX subagent interface, instead of calling directly the one of the subagent. However, this “detour” is necessary because a regular AgentX subagent cannot receive a request (Get or Set) if it has not first registered with an AgentX master agent. In order to avoid all the “administrative” steps (explained in Section 3.2.3), the RAgentX packets are received by the master agent interface, and passed onto the subagent interface.

5.2.6 Set Requests

As mentioned in Section 5.1.3, RAgentX does not make use of AgentXCommitSetPDU, AgentXUndoSetPDU, and AgentXCleanupSetPDU. The RAgentX manager only sends one Set PDU (the AgentXTestSetPDU) for each Set request, and the agent answers back with only one Response PDU.

The changes made to the NET-SNMP libraries so that the ARAX-SNMP agent would go through the phases depicted in Figure 5.2 to correctly process the RAgentX Set Requests were done in “subagent.c” C implementation file, and are shown in Figure 5.12.

When the “handle_agentx_packet” function is invoked, it checks which type of PDU has been received. If the PDU received is an RAgentXSetPDU (that is a AgentXTestSetPDU)

and the agent is acting as an RAgentX agent, the agent goes through the phases depicted in Figure 5.2 to process the Set request. The phases to be followed to process RAgentX Set requests are the same ones as the phases to process SNMP Set request, as explained in Section 5.1.3.

I have modified the code of the function “handle_agentx_packet” to specify that the RAgentX Set request should be handled as an SNMP Set request. I have done this by changing the type of the PDU to “SNMP_MSG_SET”, as shown in Figure 5.12.

```
int handle_agentx_packet(.....)
{
    .....
    switch (pdu->command) {
        .....
        case AGENTX_MSG_TESTSET:
            asi = save_set_vars(session, pdu);

            // Acting as an RAgentX
            if (netsnmp_ds_get_boolean(NETSNMP_DS_APPLICATION_ID, NETSNMP_DS_AGENT_RAGENTX_MASTER) == 1){
                asi->mode = pdu->command = SNMP_MSG_SET;
            } else //Acting as an AgentX subagent
                .....

            mycallback = handle_subagent_set_response;
            retnmagic = asi;
            break;
            .....
        default:
            return 0;
    }
    .....
    return 1;
}
```

Figure 5.12 The “handle_agentx_packet” function

5.3 RAgentX tools: Analysis and Design

As mentioned earlier, for the management of NEs of a network using a PDA, I have developed two tools: the RAgentXGet and the RAgentXSet. In the following subsections I explain in detail these tools.

5.3.1 Application Architecture

The RAgentX tools are client applications that run on PDAs and establish a connection with a server, the ARAX-SNMP agent. The purpose of the RAgentX protocol is to enable a small device to access MIB objects in the ARAX-SNMP agent, as well as any MIB object registered by AgentX subagents. As explained previously in Section 5.1, the AgentX protocol allows a small device to dynamically become a manager to manage the NEs of a network.

The RAgentX protocol includes three types of packets: get, set and response. In the implementation of RAgentX presented in this thesis, the layout of the packets is the same as in AgentX (as mentioned in Section 5.1.1). I have reused the API provided by JAX to develop the RAgentX protocol and the RAgentX tools.

In the implementation of the RAgentX tools there are two main components: The ARAX-SNMP agent and the RAgentX manager. The ARAX-SNMP agent is an RAgentX enabled NET-SNMP agent. It receives RAgentX messages through the well-known TCP port 705, processes them and returns a response. This was depicted in Figure 4.1. Figure 5.13 depicts the RAgentX tools/ ARAX-SNMP agent Model.

The NET-SNMP agent has been extended for handling RAgentX requests. These requests (queries and changes) can refer to MIB objects implemented by the SNMP agent (the static MIB) or by any AgentX subagent registered with the agent (the extended MIB).

The role of the RAgentX tools is:

- to open the connection with the ARAX-SNMP agent,
- send the requests,
- wait for the answers, and
- close the connection.

A detailed description of the tools is provided in the following sections.

Figure 5.13 RAgentX tools/ ARAX-SNMP agent Model

5.3.2 Main Classes

The main classes of RAgentX tools are:

- a) RAgentXGet: This class provides functionality to issue Get management operations using the RAgentX protocol. It opens the connection with the SNMP agent, issues the request, waits for the answer, closes the connection with the agent, and presents the received response to the manager (the PDA user).
- b) RAgentXSet: This class provides functionality to issue Set commands using the RAgentX protocol. Parallel to what RAgentXGet does, it opens the connection with the SNMP agent, issues the request with the variable and the value to be assigned (the VarBind), waits for the answer, closes the connection with the agent, and presents the received response to the manager.

These two classes, RAgentXGet and RAgentXSet, constitute the RAgentX tools.

- c) RAgentXInterface: This class is used by the RAgentX tools to send and receive RAgentX messages. It hides the transport layer details.

The following two classes, AgentXGetPDU and AgentXTestSetPDU, are part of the JAX package [31]. The RAgentX tools make use of these two PDUs classes defined for AgentX. They contain the PDU header as well as the variable binding list. Some of the parameters contained in the PDU header are the packet type, community, and the packet identifier. The packet identifier is assigned automatically.

- d) AgentXGetPDU: The RAgentXGet tool uses this class for issuing a Get request. It contains the information to be inquired from the SNMP agent.

- e) AgentXTestSetPDU: The RAgentXSet tool uses this class for issuing a Set request. It contains the variable or variables to be changed, as well as the values to be assigned to them. This information is contained in the VarBind list of the AgentXTestSetPDU class. As mentioned in Section 5.1.3, RAgentX does not make use of the AgentXCommitSetPDU, AgentXUndoSetPDU, and AgentXCleanupSetPDU classes provided by JAX for AgentX because, like SNMP, RAgentX carries out the Set request by sending only one packet.

5.4 Execution Scenarios

When the users of the PDAs start running the application, a TCP connection with the machine where the RAgentX agent is running will be open. The flow of messages between the subagent and the master agent begins as depicted in Figure 5.1. After opening the connection, the PDA acting as a manager can issue get and set requests. When the application receives the responses to the requests, it closes the TCP connection and exits.

Execution Scenario 1

Using this execution scenario I explain how the RAgentXGet tool is used. The scenario is depicted in Figure 5.14. The RAgentXGet tool was designed for requesting MIB variables. These MIB variables can be implemented either in the ARAX-SNMP agent or in AgentX subagents.

Figure 5.14 Execution Scenario 1

In this execution scenario, a PDA acting as a manager requests two MIB variables. The first one is implemented in the ARAX-SNMP agent, and the second one is implemented in another PDA (PDA 'A', that implements row 65⁷):

- a) From the ARAX-SNMP agent: the manager requests the variable that contains the contact information of the SNMP agent, that is:

iso.org.dod.internet.mgmt.mib-2.system.sysContact.0

or (in numeric representation):

.1.3.6.1.2.1.1.4.0

- b) From an AgentX subagent running on PDA 'A': the manager requests the variable that contains the OS version of the PDA, that is:

⁷ How AgentX subagents register rows in MIB tables was explained in Section 4.1.4, and an example was given in Section 4.3.

pdas.pdasMIB.pdasTable.pdasEntry.pdaOSVersion.65

or (in numeric representation):

1.3.6.1.4.1.2000.1.1.1.2.65

When the PDA user-manager starts the application (the RAgentXGet tool), a TCP connection is open with the ARAX-SNMP agent. The application creates the RAgentXGetPDU, depicted in Figure 5.15, sends the PDU to the agent, and waits for a response. The ARAX-SNMP agent answers the request by sending the RAgentXResponsePDU depicted in Figure 5.16⁸.

⁸ The numbers that are shown Figure 5.16 in v.data represent “catalina@sce.carleton.ca” and “v. 4.1”.

Figure 5.15 RAgentXGetPDU requesting two variables

Figure 5.16 RAgentXResponsePDU answering a Get request

When the application receives the answer, it presents it to the PDA user-manager, as depicted in Figure 5.17, closes the connection and exits the application.

<p>1.3.6.1.2.1.1.4.0[] = STRING: "catalina@sce.carleton.ca"</p> <p>1.3.6.1.4.1.2000.1.1.1.2.65[] = STRING: "v. 4.1"</p>

Figure 5.17 Response received from the ARAX-SNMP agent when an RAgentXGet request is issued

Execution Scenario 2

Using this execution scenario I explain how the RAgentXSet tool is used. The RAgentXSet tool was designed for setting MIB variables. These MIB variables can be implemented either in the SNMP agent or in AgentX subagents.

In this execution scenario, a PDA acting as a manager sets the MIB variable:

.1.3.6.1.2.1.1.4.0

to this value:

balita_es@yahoo.com

When the PDA user-manager starts the application (the RAgentXSet tool), a TCP connection is open with the ARAX-SNMP agent. The application creates the

RAgentXSetPDU, depicted in Figure 5.18⁹, sends it to the agent, and waits for a response. The ARAX-SNMP agent answer the request sending the RAgentXResponsePDU depicted in Figure 5.19. When the application receives the answer, it presents it to the PDA user-manager, as depicted in Figure 5.20, closes the connection and exits the application.

Figure 5.18 RAgentXSetPDU to set one variable

⁹ The numbers that are shown Figure 5.18 in v.data represent "balita_es@yahoo.com".

Figure 5.19 RAgentXResponsePDU answering a Set request

1.3.6.1.2.1.1.4.0[] = STRING: "balita_es@yahoo.com"

Figure 5.20 Result presented to the PDA user when an RAgentXSet request is issued

5.5 Modifying JAX for the RAgentX Tools

As discussed in Section 4.4, I have modified some classes provided in the JAX package to adjust to the jclPalmXtr library. Beside the changes specified in Section 4.4, for the development of the RAgentX tools several other modifications were needed.

When developing the RAgentX tools, I discovered errors in two classes provided by JAX. These errors are listed with the modifications made to the JAX package.

JAX does not implement the use of the field "prefix" in the OIDs. This field is used to reduce the length of the object identifier encodings, as mentioned in Section 3.2.2. I have implemented it to reduce the length of the packets. The modifications needed to implement this feature of the AgentX protocol are also listed below.

The modifications to the JAX classes are:

- I added a constructor to the AgentXOID class. This constructor initializes a new AgentXOID object given a String that contains the numeric representation of an OID. The constructor added is shown in Figure 5.21.

```
public AgentXOID (String input)
{
    long[] firstOID = new long[input.length()];
    int i, last = 0;
    String principio;

    if (input.startsWith(".")) input = "." + input;
    if (input.endsWith(".")) input = input + ".";

    while (input.length() > 1){
        principio = input.substring(input.indexOf('.'),input.indexOf('.',input.indexOf('.')+1));
        input = input.substring(principio.length(),input.length());
        firstOID[last]= Long.parseLong(principio.substring(1,principio.length()));
        last++;
    }

    oid = new long[last];
    for (i=0; i<last ; i++) oid[i] = firstOID[i];
    ifWithPrefix();
    setInclude(incl);
}
```

Figure 5.21 New AgentXOID constructor

- I added the attribute "prefix" in the AgentX class to represent the field "prefix" in an OID:


```
protected byte prefix = 0;
```

Two methods were also created for accessing and changing the attribute:

```
/**
 * Set the prefix field.
 */
public void setPrefix(byte num) {
    prefix = num;
}

/**
 * Get the prefix field.
 */
public byte getPrefix() {
    return prefix;
}
```

I added a method to the AgentXOID class that checks if the "internet" prefix (1.3.6.1) is part of the OID, and if so, to extract the prefix from the OID and use the field "prefix". This method is depicted in Figure 5.22.

```
/**
 * Checks if the OID has the "internet" OID as a prefix, and if so it
 * uses the field "prefix"
 */
public void ifWithPrefix(){

    long[] internet = {1,3,6,1};
    long[] aux = new long [oid.length - 5];
    boolean usePrefix = true;

    for (int count=0; count < 4; count++)
        if (internet[count] != this.oid[count])
            usePrefix = false;

    if (usePrefix==true){
        setPrefix((byte)oid[4]);
        for (int i=5,j=0; i < oid.length; i++,j++) aux[j] = this.oid[i];
        this.oid = aux;
    }
}
```

Figure 5.22 Method "ifWithPrefix()" from class AgentXOID

This method decreases the length of the packets in 20 bytes per OID that this packet has. For example, before using this message the length of an RAgentXGetPDU requesting four variables was 208 bytes. When using this method, the length is 128 bytes. (208 bytes – 128 bytes = 80 bytes = 20 bytes per OID * 4 OIDs).

- As mentioned in Section 3.2.2, the “include” field in an OID encoding indicates if the OID is included in a SearchRange. The JAX implementation never allows an OID to be included in a SearchRange. I had modified the method “getOIDBytes()” from the AgentXPDU class to allow an OID to be included in a SearchRange. The original method was:

```

/** Encode an AgentXOID with usual encoding
    @param aXOID an OID
    @return encoded OID
 */
byte[] getOIDBytes(AgentXOID aXOID)
{
    long[] oid = aXOID.getArray();
    byte[] result = new byte[(oid.length + 1) * 4];
    result[0] = (byte)oid.length;

    result[1] = 0; /* TODO implement prefix */ /* prefix field */

    result[2] = 0; /* always 0 */ /* include field */

    result[3] = 0; /* reserved */
    for(int nsid = 0; nsid < oid.length; nsid++)
        for (int i = 0; i < 4; i++)
            result[(4 * (nsid + 1)) + i] = getNthByte((int)oid[nsid], i);
    return result;
}

```

Figure 5.23 Original method “getOIDBytes()” from the AgentXPDU class

and it has been modified to:

```

byte[] getOIDBytes(AgentXOID aXOID)
{
    long[] oid = aXOID.getArray();
    byte[] result = new byte[(oid.length + 1) * 4];
    result[0] = (byte)oid.length;
    result[1] = (byte)aXOID.getPrefix(); /* prefix field */

    if (aXOID.getInclude() == true) /* include field */
        result[2] = 1;
    else
        result[2] = 0;

    result[3] = 0; /* reserved */
    for(int nsid = 0; nsid < oid.length; nsid++)
        for (int i = 0; i < 4; i++)
            result[(4 * (nsid + 1)) + i] = getNthByte((int)oid[nsid], i);
    return result;
}

```

Figure 5.24 New method “getOIDBytes()” from the AgentXPDU class

- The AgentXTestSetPDU class provided by JAX has an error in the method “getTestSetBytes”. The method “getBytes” has the purpose of converting into a byte stream the information of the class AgentXTestSetPDU. This method calls “getHeaderBytes” to encode the header, and “getTestSetBytes” to encode the rest of the PDU as follows:

```

/** encode entire PDU */
public byte[] getBytes(){
    byte[] result ;

    result=joinBytes(getHeaderBytes(),getTestSetBytes());
    return result;
}

```

The method “getTestSetBytes”, as it was originally written in the JAX package, encodes the header again into the PDU. This method was modified so that the header would not be introduced twice. The original method was:

```
private byte[] getTestSetBytes(){
    byte[] result=joinBytes(getHeaderBytes(),toVarbindBytes(tVarbind));
    return result;
}
```

and it has been modified to:

```
private byte[] getTestSetBytes(){
    return toVarbindBytes(tVarbind);
}
```

- The AgentXGetPDU class provided by JAX has an error in the method “getGetBytes”. This method is supposed to convert into a byte stream the information of the class AgentXGetPDU, encoding the OIDs to be requested consecutively. The error found was that instead of encoding the OIDs consecutively, the method encoded them one on top of each other. At the end of the encoding process only one OID was part of the AgentXGetPDU (the last one encoded), and for this only one variable requested. The original method was as follows:

```
/** Encode GetPDUs contents.
 * @return encoded contents
 */
private byte[] getGetBytes(){
    byte[] empty={ 0,0,0 , 0 };
    Enumeration enum= gSearchRange.elements();
    byte[] result =null;

    while (enum.hasMoreElements()){
        result=joinBytes(getOIDBytes((AgentXOID) enum.nextElement()),empty);
    }

    return result;
}
```

and it has been modified to:

```
private byte[] getGetBytes(){
    byte[] empty={0,0,0,0};
    Enumeration enum= gSearchRange.elements();
    byte[] result =null;

    while (enum.hasMoreElements()){
        if (result == null)
            result=joinBytes(getOIDBytes((AgentXOID) enum.nextElement()),empty);
        else {
            result=joinBytes(result,getOIDBytes((AgentXOID) enum.nextElement()));
            result=joinBytes(result,empty);
        }
    }

    return result;
}
```

Two changes were made in order to improve the presentation of the results to the user of the application. With these changes, the presentation of the results to the PDA user is more like the presentation of the results with SNMP tools:

- The method “toString” of the class AgentXVarBind gets a string representation of the VarBind. The order for printing the fields was changed in this method.

Originally the method was:

```
public String toString()
{
    String s = new String();
    s = oid.toString() + " = ";
    s = s + getValueString() + " (" + getTypeString() + ")";
    return s;
}
```

And it has been modified to:

```

public String toString()
{
    char [] aux = {};
    String s = new String(aux,0,0);
    s = old.toString() + " = ";
    s = s + getTypeString() + ":" + getValueString();
    return s;
}

```

The methods “getTypeString” and “getValueString” get the string representation of the type and the value of the VarBind respectively. Notice that the library JclPalmXtr does not provide a constructor with 0 arguments for the class String, as mentioned in Section 4.4.

```
String s = new String();
```

Each time this constructor was used in the JAX library, it was substituted with:

```

char [] aux = {};
String s = new String(aux,0,0);

```

- In the method “getTypeString” of the class AgentXVarBind the names of the VarBind’s types (Integer, String, IPAddress, etc.) were originally written in lower case. I rewrote them in capital letters (INTEGER, STRING, IPADDRESS, etc.) so that the presentation of the results would be more like the presentation provided by SNMP tools.

5.6 Development and implementation environment

The development and implementation environment of the RAgentX tools has the same characteristics as the development and implementation environment of the pdaMIB application, described in Section 4.5. Similarly to the pdaMIB application, I have been written the RAgentX tools in Java, using the JAX package [31].

For the development of the pdaMIB application I chose VAME (VisualAge Micro Edition) [27] because it provides full access to the PalmOS API. VisualAge provides libraries that allow access to low-level characteristics of PDAs. On the other hand, for the development of the RAgentX tools access to low-level information is not needed. Nevertheless, I still used this toolkit and the J9 KVM from IBM because, as it is discussed in Section 6.2, the footprint of this KVM is very small compared with other available options.

5.7 Limitations

The RAgentX tools have the same limitations as the pdaMIB application, since the pdaMIB application and the RAgentX tools have the same development and implementation environment. These limitations were described in Section 4.6. There are also some additional limitations.

Communication based on TCP

The communication between the RAgentX manager and the RAgentX agent is based on the TCP protocol. As mentioned in Section 5.6, the RAgentX tools have been developed using the VisualAge Micro Edition. This IDE provides the jclPalmXtr library for PalmOS devices. I chose this toolkit because the jclPalmXtr library allows access to low-level information in PDAs. It supports the TCP and the HTTP protocols as well. However, it does not support UDP datagrams.

In my implementation, when one of the tools is started, a TCP connection is open, and it remains open until the user exits the application. Independently of the number of messages that are sent, only one connection is open. Nevertheless, the processing time that a TCP connection needs makes the RAgentX tools slower than the SNMP tools that are based on UDP. The UDP protocol, not requiring receiving an acknowledgement for each packet it is sent, has better processing time, as is discussed in Sections 6.4.

Security

An important limitation of the RAgentX protocol is security. The RAgentX protocol aims to substitute the SNMP protocol, and make the management of a network from a resource-constrained device possible. However, RAgentX is based in AgentX. This means that there is no access control mechanism. By substituting SNMP for RAgentX, all access control is eliminated. RAgentX compromises everything, not only managed

information in the subagents, but anything that the SNMP agent has access to. A solution to this problem is proposed in Section 7.3 as part of future work.

Numeric Representation of OIDs

One thing that has not been addressed in the implementation of the RAgentX tools is the fact that the manager needs to know the numeric OID of the variables requested.

Following the philosophy of SNMP and AgentX, RAgentX messages never contain the textual representation of a variable. Variables are always requested using their numeric OID (or numeric representation). Most of the SNMP tools can work without any MIB files at all. The MIB files are used for translating between numeric and textual forms for queries and responses.

In the case of the RAgentX tools, I have not implemented a database that stores the MIB definitions in the PDA. For this reason, the PDA user cannot request MIB variables given their textual representation.

For example, if the variable that contains the contact information of a NE wants to be requested, the PDA user cannot request it by using the textual representation of the variable, that is:

iso.org.dod.internet.mgmt.mib-2.system.sysContact.0

The user needs to know the numeric OID of the variable, that is:

.1.3.6.1.2.1.1.4.0

The implementation of the RAgentX tools could be modified so that, given the textual representation of a variable, the tools could translate it into the numeric representation. In order to do this, a database with the MIB definitions should be added to the implementation.

However, there is an easier solution. The problem can be solved by presenting different options to the PDA user through the GUI (Graphical User Interface) that determines the variables requested, that is, the different options in the menus presented to the PDA user represent the different variables that can be requested. The user would not need to memorize neither the numeric OID nor the textual representation of the MIB variables, but to press buttons selecting options.

In my implementation I have used the second approach. However, I have not implemented any menu. The variables that are requested are always the same ones. I considered that the purpose of the application was to be able to manage a network using a PDA, not to program a graphical user interface (GUI). The application developed, as it is, fulfils its purpose.

5.8 Summary

At the beginning of this chapter, I described the RAgentX protocol and presented it as a protocol that can be used to manage NEs of a network from a small device. This protocol can be utilized instead of the SNMP protocol when the management is being done from a PDA. The RAgentX protocol is based on the AgentX protocol. RAgentX has the same PDUs and elements of procedure as AgentX.

I have modified the NET-SNMP agent to make it RAgentX enabled. I used the AgentX API implemented in NET-SNMP to support the RAgentX protocol. The new SNMP agent implemented is called ARAX-SNMP, and it is AgentX and RAgentX enabled.

In this chapter I have also described the characteristics of the RAgentX tools (RAgentXGet and RAgentXSet). The main classes and program architecture are presented. The RAgentX tools are client-server network applications. The tools, the clients, open a connection with the ARAX-SNMP agent, the server, to make requests. After the clients receive the responses, the connection is closed. The RAgentX tools are developed to run on PDAs with J9 VM (KVM from IBM), and written in Java using the JAX package [31].

I found errors in two of the classes provided by the JAX package. Some other changes were also needed to adjust the JAX package to the jclPalmXtr library. All these modifications are listed in Section 5.5.

In this chapter, I also discussed the applications development and implementation environment issues, as well as some limitations of the applications. The RAgentX tools share the limitations the pdaMIB application has, since the RAgentX tools and the pdaMIB application have the same development and implementation environment. However, the RAgentX tools have additional limitations. The main limitation presented is the lack of security. RAgentX does not provide any access control mechanism. Other limitation presented is that the communication in the RAgentX framework is based on TCP. This fact has an impact in the processing time of the RAgentX packets that is explained in more detail in Chapter 6.

CHAPTER 6

Results and discussion

6.1 Introduction

In this chapter I compare the footprint, bandwidth and time processing of the RAgentX tools and the pdaMIB application with SNMP. Due to the resource-constrained characteristics of some NE in UNs, the protocols used for the management of these devices need to be resource-constrained compatible. The lightness of a protocol can be defined by considering the following three characteristics: footprint, time message processing and use of bandwidth. Among these three characteristics I have stressed the first one: footprint. In a UN, the speed of message processing and the bandwidth used are important. However, the most critical factor of the resource-constrained devices, and thus my main focus, is the footprint.

PDA's are resource-constrained, mobile, and communication capable devices. These are the same characteristics with which I had identified devices that might form UNs in Section 1.1. For this, PDA's are very good representatives of resource-constrained NEs of UNs.

As mentioned in Section 1.2, the motivation for the development of the pdaMIB application was the need to dynamically extend the SNMP agents' MIB. The pdaMIB application is an AgentX subagent running on a PDA. This application allows PDA's to

dynamically register the MIB region they implement, and thus dynamically extend the SNMP agents' MIB.

The RAgentX tools have been also developed to run on PDAs. These tools allow PDAs to act as managers, to request and change values of MIB variables. The RAgentX protocol can be used instead of the SNMP protocol to manage networks from a small device. For this reason, I compared RAgentX with SNMPv1 and SNMPv3. Even though SNMPv3 is less lightweight than SNMPv1 in terms of use of bandwidth, I considered that SNMPv3 has recently become the standard, so it is now the SNMP version to use.

As mentioned in Section 3.1, SNMPv3 has three levels of security. I have compared RAgentX with SNMPv1, and SNMPv3 with the three levels of security: SNMPv3 without authentication and privacy (noAuthNoPriv), SNMPv3 with authentication and no privacy (authNoPriv), and SNMPv3 with authentication and privacy (authPriv). On the other hand, RAgentX does not provide any access control mechanism, as I mentioned in Section 5.7. I focused on the comparison of RAgentX with SNMPv1 and SNMPv3 without authentication and privacy (noAuthNoPriv) because the mechanisms provided by SNMPv1 and SNMPv3 noAuthNoPriv are not enough for full security and are therefore more comparable to RAgentX.

The roadmap of this chapter is as follows. In Section 6.2 I present the results of the tests executed to measure the footprint of the applications developed. In Section 6.3 I describe the tests I have carried out to measure the bandwidth use and I present the results of these

tests. Section 6.4 presents the tests executed to measure processing time and their results. In Section 6.5 I discuss the results obtained in the tests of the previous sections. Finally, Section 6.6 presents a summary of this chapter.

6.2 Footprint

The term “footprint” of an application refers to the amount of memory space a particular application occupies. In this section I distinguish between the amount of disk space that a piece of software occupies when idle, and the memory they use when running. I refer to the first concept as “disk space occupation” (DSO), and I refer to the latter concept as “footprint”.

6.2.1 Management of a NE

For the management of a NE in a UN, in the SNMP/AgentX framework, there are several possible solutions. These solutions are:

- 1) to run an SNMP agent on the NE
- 2) to run an AgentX subagent on the NE

The candidates for the option 1) are: a) the original NET-SNMP agent and b) the NET-SNMP agent with minimum features (hereafter referred as “min. features”). I have named the first one “original” NET-SNMP because I have not made any modifications to it. The latter one is the original NET-SNMP agent stripped of all unnecessary features.

As mentioned in Section 5.2, the NET-SNMP agent provides not only a built-in AgentX master agent, but also an AgentX subagent. The two candidates for the option 1) can act either as an AgentX master agent (AMA), or as an AgentX subagent (AS). The footprint and the disk space occupation differ depending on the role. For this reason, the footprint and the disk space occupation are calculated and presented separately for each of the roles these applications take. For example, the footprint of the original NET-SNMP when acting as an AgentX master agent (AMA) is 6.6 MB, while the footprint of the original NET-SNMP when acting as an AgentX subagent is 4.4 MB.

However, the disk space occupation of the original NET-SNMP agent acting as an AgentX master agent (AMA) is the same (2.65 MB) as acting as a subagent (AS) because, as mention above, the NET-SNMP agent has a built-in an AgentX master agent and an AgentX subagent. The disk space occupation (because it refers to the agent when it is not started) does not reflect in this case the different roles that will be reflected in the footprint when the agent is running. This is shown in Table 6.1.

The candidates for option 2) are: a) the AgentX subagent I have developed to run on PDAs (the pdaMIB application), and b) an AgentX subagent I have developed to run on a regular JVM. The latter option was developed as an intermediate step in the development of the pdaMIB application. It does not implement a MIB, but I have used it as a representative example of AgentX subagents implemented in Java (SDK 1.4.0_01). The pdaMIB application, as it has been mentioned before, runs on J9 VM (the KVM from IBM).

Not all the options presented above are feasible solutions for the resource-constrained devices in a UN. Some of the options presented are not resource-constrained compatible. A more detail explanation about this issue is given below.

Table 6.1 and Table 6.2 show the disk space occupation (DSO) comparison. In Table 6.1 the amounts of DSO presented do not include the amount of DSO that the environment or the virtual machine (VM) require. The DSO of both, the VM and the environment, are listed as well. Table 6.2 presents the total amount of DSO that the different options require. This table has been included to facilitate the comparison between the different options.

	AgentX master agent (AMA)	AgentX subagent (AS)	VM or environment
NET-SNMP “original”	2.65 MB	2.65 MB	2.15 MB
NET-SNMP “min. features”	1.53 MB	1.46 MB	2.06 MB
Application on JVM	N/A	100.15 KB	4 MB
Application on J9 VM (KVM)	N/A	83.3 KB	223 KB

Table 6.1 Agents/subagents Disk Space Occupation (DSO) Comparison

In Tables 6.1 and Table 6.3 “N/A” stands for “Not applicable”. It means that no software has been developed for that scenario. For example, I have not developed an AgentX master agent (AMA) to run either on a regular JVM or on a J9 VM.

	AMA with “original”	AS with “original”	AMA with “min. features”	AS with “min. features”	AS on JVM	AS on J9 VM
Total amount of DSO	4.8 MB	4.8 MB	3.59 MB	3.52 MB	4.09 MB	306.3 KB

Table 6.2 Agents/subagents Total Amount of DSO Comparison

As shown in Table 6.1, the NET-SNMP AgentX subagent stripped of all unnecessary features (“min. features”) does not have the smallest disk space occupation. This subagent needs 1.46 MB of disk. Furthermore, the amount of disk space that the environment occupies (2.06 MB) has to be added to the amount of 1.46MB. Consequently, the total amount of disk space required for this option is 3.52 MB, as shown in Table 6.2. Nevertheless, this option needs less total disk space than the AgentX subagent running on a regular JVM that requires approximately 4.09MB (JVM included). This is due to the fact that the DSO required for the JVM is larger than the DSO required for the environment of the “original”.

As it can be deduced from the example above, in order to have a small DSO not only the application’s DSO needs to be small, but also the DSO of environment or VM needs to be small.

On the other hand, the pdamIB application occupies 83.3 KB of disk space (plus the 223 KB of disk space needed for the J9 VM). The total amount of disk space required for the

application developed in this thesis for the management of resource-constrained devices is **306.3 KB**, as shown in Table 6.2.

Table 6.3 and Table 6.4 show the Agents/subagents footprint comparison. Table 6.3 presents the footprint of the different options without including the footprint of the VM or the environment, while Table 6.4 shows the footprint of the options including the environment or VM.

	AgentX master agent (AMA)	AgentX subagent (AS)	VM or environment
NET-SNMP “original”	6.6 MB	4.4 MB	2.15 MB
NET-SNMP “min. features”	3.83 MB	3.66 MB	2.06 MB
Application on JVM	N/A	788 KB	6.8 MB
Application on J9 VM (KVM)	N/A	452 KB ¹⁰	N/A ¹⁰

Table 6.3 Agents/subagents Footprint Comparison

¹⁰ The J9 VM footprint is included in the footprint of the AgentX subagent.

	AMA with “original”	AS with “original”	AMA with “min. features”	AS with “min. features”	AS on JVM	AS on J9 VM
Total amount of footprint	8.75 MB	6.55 MB	5.89 MB	5.72 MB	7.56 MB	452 KB

Table 6.4 Agents/subagents Total Amount of Footprint Comparison

The NET-SNMP “min. features” AgentX subagent (AS with “min. features”) has a footprint of 3.66 MB. The footprint of the environment is 2.06 MB. Consequently, the total footprint of this option is 5.72 MB, as shown in Table 6.4. Other options, like installing the original NET-SNMP AgentX master agent in the device to be managed (8.75 MB), have larger footprints. The pdmMIB application has the total footprint of **452 KB**.

As mentioned in Section 2.2.1, PDAs are generally provided with 8 or 16 MB of RAM and 4 MB of ROM. From here on, I have taken the worst-case scenario and I have considered that all PDAs have 8 MB of RAM. Even though this might not be true for all PDAs, it might be the case of many resource-constrained devices in a UN. As mentioned before, I have chosen the PDAs as representatives of resource-constrained devices.

PDAs employ the RAM to store the operating system, standard applications and additional programs. A Palm IIIc handheld with 8MB of RAM uses 22 KB of the RAM

to store the OS and 0.3 MB to store the standard applications that come with any PalmOS device. Thus, the total amount of “free” RAM to store other applications is 7.7 MB.

In a PDA with 8 MB of RAM, such as a Palm IIIc handheld, the pdaMIB application makes little use of the RAM; it only uses 306.3 KB of the RAM to be stored, leaving the rest of the memory available to store other applications. On the other hand, the other options presented in Table 6.2 use about half of the RAM available in the 8 MB RAM PDA merely to store the application and the environment or the VM. This is the case, for example, of the AgentX subagent running on a regular JVM that requires approximately 4.09MB of RAM.

The pdaMIB application disk space occupation is approximately 11-16 times smaller than for the other options. Its footprint is also smaller. It is approximately 12-20 times smaller than the other options.

From the point of view of the DSO, all the options presented above are feasible as for all the applications and their environment or their VM can be stored in the RAM of an 8 MB RAM PDA (because their DSO is smaller than 7.7MB¹¹). However, all the options, but the pdaMIB application, occupy about half of the RAM of the PDA not leaving much memory space available for other applications.

¹¹ 7.7MB is the amount of RAM memory available in an 8 MB RAM PDA. The other 0.3 MB are used by the OS and the standard applications.

It is important to stress that the main purpose of a PDA is not to be managed, but rather to facilitate the work of the user. This is done by the applications installed on the PDA. The fact that the software installed to manage a PDA occupies half of its RAM makes difficult to have other applications on the PDA, and thus to fulfil the PDA's purpose. For this reason, I do not consider the above options, but the pdaMIB application, feasible to be used for the management of small devices.

From the point of view of the footprint, all solutions presented above are also feasible as for when they are running these applications do not exceed the RAM available in an 8 MB RAM PDA. However, all the options, but the pdaMIB application, have a footprint that requires half of the RAM, leaving little memory space available for other applications. This again defeats the purpose of the PDA.

In summary, even though all the possible solutions presented above can be stored and can be run on a 8 MB RAM PDA, I do not consider them feasible solutions for the management of resource-constrained devices in a UN because they require about half of the RAM of the PDA. Only the pdaMIB application occupies so little space of the RAM that allows many other applications to be stored and run.

6.2.2 Management from a NE

For the management of UNs from a NE, in the SNMP/AgentX framework, there are also two solutions:

- 1) to use the SNMP tools,

2) to use the RAgentX tools.

The candidates for option 1) are: a) the “original” NET-SNMP package and b) the NET-SNMP with minimum features (“min. features”). The NET-SNMP package provides a group of tools for network management, such as the SNMP Get tool and the SNMP Set tool. NET-SNMP “min. features” provides the same tools.

The candidates for option 2) are: a) the RAgentX tools I have developed to run on PDAs, and b) the RAgentX tools I have developed to run on a regular JVM. The latter option was developed as an intermediate step in the development of the RAgentX tools. I have used them as a representative example of RAgentX tools implemented in Java (SDK 1.4.0_01). The RAgentX tools, as it has been mentioned before, runs on J9 VM (the KVM of IBM).

However, not all the options presented above are feasible solutions for resource-constrained devices in a UN. Many of the options presented are not resource-constrained compatible because they require more memory than it is available in some PDAs.

Table 6.5 and Table 6.6 show the disk space occupation (DSO) comparison of these management tools. In Table 6.5 the amounts of DSO presented do not include the amount of DSO that the environment or the virtual machine (VM) require. In Table 6.6 the total amount of DSO presented do include the DSO of the VM and the environment. Table 6.7 presents the footprint comparison of the different options of management tools without

including the environment or the VM, while Table 6.8 presents the footprint of the applications including the environment or the VM.

	Get tool	Set tool	VM or environment
NET-SNMP “original”	898 KB	900 KB	2.15 MB
NET-SNMP “min. features”	847 KB	848 KB	2.06 MB
Application on JVM	92.27 KB	93.09 KB	4 MB
Application on J9 VM (KVM)	60 KB	63 KB	223 KB

Table 6.5 Management Tools Disk Space Occupation Comparison

	“original”	“min. features”	JVM	J9 VM
Get tool	3.02 MB	2.88 MB	4.09 MB	283 KB
Set tool	3.02 MB	2.88 MB	4.09 MB	286 KB

Table 6.6 Management Tools Total amount of DSO Comparison

As shown in Table 6.6, the NET-SNMP “min. features” Get tool requires a total amount of disk space of 2.88 MB (including the environment). Nevertheless, this option needs less disk space than the Get tool running on a regular JVM that requires approximately 4.09MB (JVM included). On the other hand, the RAgentXGet tool occupies 283 KB of disk space (J9 VM included).

The same occurs with the footprint. The RAgentXGet and RAgentXSet tools have a much smaller footprint than the other options presented in Table 6.8. The RAgentXGet tool has the footprint of 375 KB, while the smallest footprint of the other possible Get tool options is 4.19 MB. This is the footprint of the NET-SNMP “min. features” (including the environment).

	Get tool	Set tool	VM or environment
NET-SNMP (original)	3.3 MB	3.51 MB	2.15 MB
NET-SNMP (min. features)	2.13 MB	2.58 MB	2.06 MB
Application on JVM	756 KB	760 KB	6.8 MB
Application on J9 VM (KVM)	375 KB	388 KB ¹²	N/A ¹²

Table 6.7 Management Tools Footprint Comparison

	“original”	“min. features”	JVM	J9 VM
Get tool	5.45 MB	4.19 MB	7.54 MB	375 KB
Set tool	5.66 MB	4.64 MB	7.54 MB	388 MB

Table 6.8 Management Tools Total Amount of Footprint Comparison

The RAgentXGet and RAgentXSet tools occupy little memory space of the RAM, leaving the rest of the RAM free to store other applications. For a PDA with 8 MB of

¹² The J9 VM footprint is included in the footprint of the RAgentXGet and RAgentXSet tools.

RAM the RAgentX tools are feasible solutions for the management of a network from a PDA. The RAgentXGet and RAgentXSet tools, as mentioned above, require 283 KB and 286 KB of disk space respectively.

Other options presented in Table 6.8 use for running more than half of the RAM available in a PDA. For example, the Get tool provided by NET-SNMP “original” has a footprint of 5.45 MB and the Set tool running on a regular JVM requires 7.54 MB.

As mentioned before, there are 7.7 MB of RAM available in an 8 MB RAM PDA to installed applications. If a Get tool application that runs on a regular JVM with a footprint of 7.54 MB is installed and run in a PDA, only an application of 164 KB of DSO can be installed in the PDA.

As mentioned in Section 6.2.1, the main purpose of a PDA is to provide the user with applications that facilitate his work. The fact that an application requires a large amount of RAM not allowing other applications to be installed defeats the purpose of the PDAs. Furthermore, the bigger the footprint and DSO of an application, the less portable this application is to resource-constrained devices.

The RAgentX tools disk space occupation is approximately 10-14 times smaller than for the other options. Its footprint is also smaller. It is approximately 11-20 times smaller than for the other options.

In summary, all the options presented above can be installed and run on a PDA with 8 MB of RAM. However, only the RAgentX tools' footprint and DSO are so small that do not defeat the purpose of the PDAs and allow many other applications to be installed and run.

6.3 Bandwidth Use

The length of the packets sent for the communication between the SNMP agents and the managers (using either SNMP or RAgentX) determines the bandwidth used by the protocols.

6.3.1 Test Cases

To compare the bandwidth used by SNMP, AgentX and RAgentX I have run a group of tests. These tests measure packets' length sent between the managers and the SNMP agents or AgentX subagents. In these tests there are two dimensions to take into consideration: 1) which variable is requested, and 2) which tool requests the variable (the SNMP tools or the RAgentX tools). For these tests the entities' location is not relevant. In other words, the length of the packets does not vary depending on which machine the entities are running.

There are four tools to test: a) the RAgentXGet tool, b) the SNMP Get tool, c) the RAgentXSet tool, and d) the SNMP Set tool. The tools are tested in pairs: 1) the former two tools, and 2) the latter two tools.

There are two types of variables to be requested: a) variables that are implemented in the SNMP agent and are in the static MIB, and b) variables that are implemented by an AgentX subagent and are in the dynamic MIB.

Additionally, to better compare the bandwidth consumed by each one of the protocols, the VarBind lists of the requests are of different length. I have compared the length of the packets when requesting one variable and four variables. These test cases are summarized in Table 6.9.

Test Case	Test 1	Test 2	Test 3	Test 4
VarBind	from static MIB	from static MIB	from dynamic MIB	from dynamic MIB
VarBind List Length	1	4	1	4

Table 6.9 Test Case Length Comparison

6.3.2 Packet's Length Comparison

There are several factors to take into account when reading the tables that contain the comparison of the packets' length:

- When a manager issues a request, it sends a packet (a Get or Set request) that is answered by the SNMP agent with another packet (a response). The lengths contained in Table 6.10 and Table 6.11 are the addition of the lengths of these two packets. I refer to these lengths as "total length". These two packets were depicted in Figure 5.1 (for RAgentX) and Figure 3.8 (for SNMP).

	Test 1	Test 2	Test 3	Test 4
SNMPv1	110	222	108	244
SNMPv3 NoAuthNoPriv	249	361	246	385
RAgentX	136	320	144	372
AgentX	N/A	N/A	152	440
SNMPv3 AuthNoPriv	273	386	272	407
SNMPv3 AuthPriv	308	413	300	438

Table 6.10 SNMP, AgentX, and RAgentX GET packets. Length comparison (bytes)

- In SNMPv3 there is overhead. Before issuing a request (Get or Set), the manager has to ask for the Engine number from the SNMP agent. This exchange of messages is part of SNMPv3's replay attack protection and is also needed for key localization. In NET-SNMP [32], the implementation of SNMP used in this thesis, the overhead is of 173 bytes per request issued. However, in other SNMP implementations, like Westhawk [36], the overhead happens only once. In Westhawk, before issuing a group of requests (Get or Set), the manager asks for the Engine number from the SNMP agent the manager. The manager stores the Engine number in order to use it in the requests to be issued. That is, instead of asking for the Engine number before issuing each request, the manager asks for the Engine number at the beginning, stores it, and uses it in each request.

- Test 1 and Test 2 are not applicable (N/A) to AgentX because in these tests I request variables from the static MIB, and so the AgentX protocol is not used.

	Test 1	Test 2	Test 3	Test 4
SNMPv1	126	246	110	252
SNMPv3 NoAuthNoPriv	264	384	246	392
RAgentX	108	212	112	260
AgentX	N/A	N/A	168	316
SNMPv3 AuthNoPriv	288	408	272	416
SNMPv3 AuthPriv	316	430	300	446

Table 6.11 SNMP, AgentX, and RAgentX SET packets. Length comparison (bytes)

Get Requests Comparison

Table 6.10 shows the SNMP, AgentX and RAgentX GET packets' length comparison.

This table shows that the RAgentX packets are shorter than SNMPv3 packets, but longer than SNMPv1 packets.

The SNMPv3 packets contain some fields, such as securityLevel, securityModel, msgSecurityParameters, etc. [38], that are used to specify security information. These

fields are always present in the header of the SNMPv3 packets. SNMPv1 only has a field to specify security. This field contains the community name.

On the other hand, the RAgentX's packets have only one field to specify the context. Furthermore, this field is optional (as it was explained in Section 3.2.2 when describing the AgentXGetPDU). If no context is specified, the field to indicate the context is not present in the packet.

RAgentX packets are shorter than SNMPv3 packets because the latter have some fields that are used for containing security information while the former do not.

In the AgentX and RAgentX protocols, the length of the packets depends on the variables requested. More precisely, these protocols packets' length depends very much on the numeric OID of the variable requested.

To simplify the explanation that follows, I have called the variables that are implemented in the SNMP agent and are in the static MIB "static variables". On the other hand, I have called "dynamic variables" those variables implemented by the AgentX subagents and that are in the dynamic MIB.

The total length of the RAgentXGet packets when requesting one "static variable" is 136 bytes. On the other hand, if the variable requested is a "dynamic variable", the total length of the packets is 144 bytes.

As mentioned earlier, the lengths contained in Tables 6.10 and Table 6.11 are the addition of the length of the request and the response packets exchanged between the manager and the SNMP agent. The length of the request packet sent when requesting a “static variable” is 56 bytes, while the length of the request packet sent when requesting a “dynamic variable” is 68 bytes. The 12 bytes difference between the two lengths shows that the numeric OID that represents the “dynamic variable” (1.3.6.1.4.1.2000.1.1.1.2.65) is longer than the numeric OID of the “static variable” (1.3.6.1.2.1.1.4.0). In fact, the numeric OID of the “dynamic variable” has 3 sub-identifiers more than the numeric OID of the “static variable”. Each sub-identifier in AgentX is encoded using 4 bytes. Hence, there are 12 extra bytes in the latter RAgentXGet packet (3 “extra” sub-identifiers * 4 bytes per sub-identifier).

The difference in the packets’ total length is greater if the number of variables requested increases. The total length of a RAgentXGet packet when requesting four “static variables” is 320 bytes. The total length of the packet when requesting four “dynamic variables” is 372 bytes. Before the difference between the packets’ total length was 8 bytes. Now it has increased to 52 bytes.

These two examples have been explained to highlight how the numeric OID of the variable requested affects the RAgentX packet’s length. A similar occurrence to that described above is found in the AgentX packets’ length.

However, the length of the numeric OIDs of variables requested do not affect the SNMP protocol as much as it does the RAgentX and AgentX protocols. The total length of the packets to request one “static variable” is 110 bytes (with SNMPv1) and 249 bytes (with SNMPv3 noAuthNoPriv). The total length of the packets to request one “dynamic variable” is 108 bytes (with SNMPv1) and 246 bytes (with SNMPv3 noAuthNoPriv). This shows that the total length of the packets does not vary very much. The SNMP packets’ length is not completely independent of the variables requested; however, the length of the numeric OID of the variable requested does not affect SNMP packets’ length as much as it does the RAgentX and AgentX packets’ length.

Set Requests Comparison

Table 6.11 shows the SNMP, AgentX and RAgentX SET packets’ length comparison. This table shows that the RAgentX packets are shorter than SNMPv3 packets. RAgentX packets are also shorter than SNMPv1 when requesting one variable, and only a little longer when requesting more than one.

In general, RAgentX packets used for setting variables are shorter because the responses sent by the RAgentX agent to the manager only contain the AgentX header and 3 fields in which it is specified if an error has occurred. This type of AgentXResponsePDU was depicted in Figure 3.6. Independently from the success or failure of the setting process, the RAgentX agent always sends 28 byte packet responses to the RAgentX manager.

As explained in the subsection above, SNMPv3 packet's header contains fields to specify security information. The RAgentX packets only contain one optional field to specify the context. This explains why RAgentX packets are shorter than SNMPv3 packets.

When setting variables implemented by the AgentX subagent, the AgentX master agent and the subagent exchange five packets (if the setting process is successful). These five packets were depicted in Figure 3.9. In Table 6.11 the sum of the lengths of these packets is presented.

In summary, the bandwidth used by the RAgentX protocol is lower than the other compared protocols, with exception of SNMPv1 in some scenarios. However, SNMPv1 and SNMPv3 packets' length do not depend on the length of the numeric OID of the variables requested, and RAgentX packets' length do. Nevertheless, RAgentX's bandwidth use is lightweight enough for the management of a network from resource-constrained devices.

6.4 Processing Time

I have run a group of tests to measure the response time of the RAgentX and SNMP requests sent to the ARAX-SNMP agent. In these tests, the ARAX-SNMP agent and the management applications are running on the same machine.

In order to obtain more comparable results, I have varied the number of replications and the length of the request's variable binding list the management tools send to the ARAX-

SNMP agent. Timestamps were taken in every replication, each time recording the response time and finally calculating the average. Both tools, SNMPGet and RAgentXGet, calculate and print out the average response time as result. These test cases are summarized in Table 6.12.

Test Case	Test 1	Test 2	Test 3	Test 4	Test 5	Test 8
Replications	500	1000	500	1000	500	1000
VarBind List Length	1	1	4	4	10	10

Table 6.12 Processing Time Test Case

6.4.1 Test Configuration

Figure 6.1 shows the test configuration. The ARAX-SNMP agent, the AgentX subagent, the RAgentX and SNMP management tools are all executing on INM-01.

INM-01 is a 1400MHz Windows 2000 machine, with 256 MB of RAM. The ARAX-SNMP agent is a modified version of the NET-SNMP agent, written in C and compiled on a windows machine with Cygwin (UNIX environment for Windows). The ARAX-SNMP agent receives SNMP, AgentX and RAgentX messages from the well-known TCP ports 161 and 705, as depicted in Figure 6.1. The SNMP tools are running under Cygwin as well.

For these tests I have used a Palm m505 handheld emulator (16 MB RAM, 8 MB ROM, Palm OS 4.1, 33 MHz processor speed) to run the RAgentXGet tool as well as the

pdaMIB application. However, the emulator (33 MHz processor speed) is very slow in comparison with INM-01 (1400MHz processor speed). To have a consistent comparison between SNMP and RAgentX, I have also tested the RAgentXGet tool running on a regular JVM on INM-01. I refer to the latter scenario as a “RAgentX tools”, and to the former one as “RAgentX tools on a PDA emulator”.

Figure 6.1 Test Configuration

The RAgentXGet tool is an application written in Java that measures the time since a request is sent to the ARAX-SNMP agent, until a response is received. The communication between the RAgentXGet tool and the SNMP agent is based on TCP. To get a more comparable result, the TCP connection between the two entities is open only once. Once all the tests have finished executing this connection is closed.

The SNMP Get tool is based on UDP and written in C. I have modified the tool provided by NET-SNMP so that it would measure the response time of Get requests.

6.4.2 Processing Time Comparison

The tests were executed, and Table 6.13 shows the average processing time comparison.

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
SNMPv1	14.04	14.05	14.05	14.08	13.86	13.74
SNMPv3 NoAuthNoPriv	10.35	10.37	10.65	10.69	11.24	10.85
RAgentX	15	17	20	20	21	20
SNMPv3 AuthNoPriv	10.84	10.59	11.19	10.95	11.04	11.28
SNMPv3 AuthPriv	11.17	10.78	11.61	12.50	12.57	11.89
RAgentX on a PDA emulator	94	94	206	206	470	470

Table 6.13 Average Processing Time (msec)

The fastest protocol is SNMPv3 without authentication and privacy (NoAuthNoPriv), followed by SNMPv3 with authentication and no privacy (authNoPriv). The RAgentX request average response time is, in all these test cases, approximately 1.4-1.8 times slower than the fastest version of SNMP (SNMPv3 NoAuthNoPriv), and approximately 1-1.5 times slower than the slowest of the SNMP versions (SNMPv1). Nevertheless, RAgentX is in the same order of magnitude as all the SNMP versions.

The explanation of the difference in processing time between the SNMP and RAgentX protocols can be found in the fact that the latter is based on TCP, and the first one is based on UDP. Section 6.5 presents a discussion of the impact that the transport protocol has.

Table 6.13 also shows the average processing time of the RAgentX protocol when the RAgentX tools are running on a PDA emulator, as mentioned in Section 6.4.1. As expected, the processing time in this scenario is much greater because the processor speed of the emulator is only 33MHz.

In summary, the processing time of the RAgentX protocol is greater than the other compared protocols. However, the difference between RAgentX and SNMPv1 processing time is in some scenarios very small. Nevertheless, the RAgentX processing time is low enough for the management of a network from small devices.

6.5 Discussion

The pdaMIB application and the RAgentX tools developed in this thesis are based on TCP. This is a very influential factor that is detrimental to time processing. TCP connections need more processing time because each one of the messages exchanged between the connection endpoints is acknowledged.

The applications I have implemented are based on TCP because the library used for their development does not support UDP. However, I chose to use that library for three reasons: it provides access to low-level information of PDAs, has a very small footprint KVM, and allows the development of small footprint applications. For the development of the pdaMIB application the access to low-level information is fundamental. The KVM used for running the applications I have developed has a smaller disk space occupation than other KVMs, for example Java HQ, the KVM from the Wireless toolkit (WTK) of Sun [37]. The disk space occupation of the Java HQ is 606 KB, while the disk space occupation of J9 VM is 223 KB.

The pdaMIB application and RAgentX tools have a much smaller footprint and disk space occupation than the other options presented as possible solutions a) for the management of resource-constrained devices, and b) for management of networks from resource-constrained devices. The applications I have developed are 10-20 times smaller than the other presented options. Thus, they leave more free memory space than the other options for the PDA user to install other applications.

On the other hand, the RAgentX tools have greater processing time than the other presented options. Their bandwidth use is lower than the other compared protocols, with exception of SNMPv1 in some scenarios.

Some ideas have been proposed in order to improve the processing time and the bandwidth use of the RAgentX protocol. These ideas are gathered in Section 7.3.

In summary, the applications I have developed require more processing time and bandwidth than the other options presented as possible solutions. However, the footprint and the disk space occupation of these applications compensate the downside in processing time and bandwidth use.

6.6 Summary

In this chapter I have presented the results obtained from the tests done with the AgentX and the RAgentX protocol. I provided a comparison between SNMP and RAgentX to arrive to the conclusion that RAgentX is a suitable protocol when the management of a network is done from a small device, like a PDA.

In this chapter I have presented results to prove that the AgentX subagent (the pdaMIB application) and the RAgentX tools I have developed are sufficiently lightweight and resource-constrained compatible. I approached this by showing the footprint, the time for message processing and the use of bandwidth of these two protocols.

CHAPTER 7

Conclusions and Future Work

7.1 Conclusions

Ubiquitous networks (UN) are those that have computers and computation everywhere, embedded in walls, tabletops, and other everyday objects [8]. When I started my thesis, no methodology had yet been established for the management of UNs.

Some devices in this type of networks are resource-constrained, capable-of-communication and mobile. Given these characteristics, it is necessary to have the capacity for dynamic incorporation of devices and their information management into the network.

The SNMP protocol is the protocol most widely used in network management systems (NMS) [2]. Nonetheless, the SNMP framework does not provide a mechanism for dynamically extending the SNMP MIB; whereas, the AgentX protocol (part of the SNMP family) does.

The AgentX protocol was originally designed for extending the SNMP MIB with both the AgentX master agent and the AgentX subagents running on the same machine. I have broadened the concept and framework this protocol provides in order to use it in wired and wireless networks.

In this thesis, I have developed an AgentX subagent, the pdaMIB application. This application runs on a PDA, and permits the PDA user to dynamically register the MIB implemented in the PDA with an SNMP agent. I have also developed two tools that allow a PDA user to manage a network from a PDA. The protocol defined and implemented for this purpose is the RAgentX protocol.

If the framework developed in this thesis were to be applied to the UN formed in a household, one would have one ARAX-SNMP agent and AgentX subagents running on all the resource-constrained everyday appliances to be managed that register their MIBs with the ARAX-SNMP agent. The pdaMIB application is the prototype of these AgentX subagents.

In Chapter 6, I compared the applications developed in this thesis with other possible solutions. As the other solutions do not have a small footprint, they failed to fulfill one of the most important demands of the resource-constrained devices in UNs.

The RAgentX tools and the AgentX subagent pdaMIB application are sufficiently lightweight to operate on a mobile and wireless communicated device with limited resources. This application and these tools provide a feasible solution for the management of Ubiquitous Networks.

7.2 Thesis contributions

The main contributions of this Master's thesis are:

a) I have developed the pdaMIB application. This application is an AgentX subagent that allows the dynamic extension of the SNMP agent MIB by registering its MIB region. Although this application runs only on PDAs, it is a prototype for the management of resource-constrained devices. The pdaMIB application was explained in Chapter 4.

b) I have defined a PDA MIB that is implemented by the pdaMIB application. This MIB collects useful information for the manager of a UN, such as version of the OS, number of databases in the PDA, percentage of power remaining in the battery of the PDA, etc. The PDA MIB definition was presented in Section 4.2.

c) I have modified and corrected some errors in the JAX package [31] in order to develop the pdaMIB application. The JAX package provides an API for the deployment of AgentX subagents. Some classes of the JAX package have been modified to adjust them to the jclPalmXtr library¹³. The modifications and errors found were explained in Section 4.4 and Section 5.5.

d) I have developed two management tools to manage a network from resource-constrained devices. These tools, the RAgentXGet and RAgentXSet, run on PDAs as well as on any other device with a JVM.

e) I have defined and implemented a new protocol, the RAgentX protocol, for the communication between the RAgentX tools and the SNMP agent. This protocol is based

¹³ Java class library that has been custom-designed for developing software for Palm OS devices.

on AgentX, and can be used instead of SNMP for the network management from a small device. RAgentX is a resource-constrained compatible protocol. A detailed explanation of the lightness of RAgentX was provided in Chapter 6.

f) I have modified the SNMP agent provided by the NET-SNMP package [32] to make it RAgentX enabled. The NET-SNMP implementation already provides an AgentX enabled SNMP agent. The new implementation is called ARAX (AgentX RAgentX) – SNMP agent.

g) Overall in this thesis, I have provided a feasible solution for the management of UNs. This solution consists of having one ARAX-SNMP per household, and an AgentX subagent on each managed small device in the UN.

7.3 Future Work

During this research work, a number of directions have been identified for future work.

a) A very important issue not addressed in this thesis is security. As mentioned in Section 5.7, the RAgentX protocol does not provide any access control mechanism. Because RAgentX can be used instead of SNMP, it compromises everything, not only the managed information in subagents, but anything that the SNMP agent has access to. However, the RAgentX protocol has in its PDU's header a field designed to specify a context. The protocol could be modified to make use of this field to provide some kind of security.

b) RAgentX could be altered to be more lightweight. In the RAgentX protocol, when the manager issues a request the OIDs of the variables requested are contained in both the request and the response PDUs. However, the variables' OIDs are not needed in the response PDU sent by the RAgentX agent to the manager because the manager knows which variables were requested. To make the protocol more lightweight, only the variables' OIDs could be sent in the requests, and only the variables' values could be sent in the responses. Doing this, repeating unnecessary information is avoided.

c) The solution presented in this thesis as feasible solution of UN, makes use of the AgentX framework. The number of AgentX subagents registered with an SNMP agent is limited by the maximum number of open file descriptors that the system supports. This could be a problem if the number of devices in a UN is very high. In such a case, other solutions should be provided.

d) The RAgentX tools are based on TCP. This has some drawbacks in term of time processing and bandwidth use. In the future, these tools could be developed using UDP. However, for this the jclPalmXtr library cannot be used.

e) As pointed out in Section 5.7, the RAgentX tools cannot request MIB variables using their textual representation. In order to allow this, a database that stores the PDA MIB definitions needs to be implemented.

f) In this thesis two tools have been developed: RAgentXGet and RAgentXSet. In the future, other tools, such as RAgentXGetNext, could be developed.

References

- [1] Stalling W. SNMP, SNMPv2, SNMPv3, and RMON 1 and 2. Third Edition. Addison-Wesley Longman, 1999.
- [2] Subramanian, M., Network Management: Principles and Practice. Addison-Wesley, 2000.
- [3] Kumar D., et al. End-to-End Management with Tivoli: Managing PDAs. IBM Redbook, May 2000. [Online document]. Available URL: (last time visited May 6th, 2003)
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246001.pdf>
- [4] Hoffman R. Living in Nomad's Land: Managing Mobile Devices. Network Computing Magazine, July 2000.
- [5] Ariston Catalogue Web Page. [Online document]. Available URL: (last time visited May 6th, 2003)
<http://www.merloni.com/eng/index.jhtml>
- [6] Want R., Schilit B.N., Adams N.I., Gold R., Petersen K., Goldberg D., Ellis J.R., Weiser M. The PARCTAB Ubiquitous Computing Experiment. Technical Report CSL-95-1, Xerox Palo Alto Research Center, March 1995.
- [7] Weiser M. The computer for the Twenty-first century. Scientific American, 265 No.3, pp. 94-104, September 1991.
- [8] Weiser M., Brown J.S. The coming Age of Calm Technology. Xerox PARC. October 1996. [Online document]. Available URL: (last time visited May 6th, 2003)
<http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm>

- [9] Weiser M. Some Computer Science Issues in Ubiquitous Computing. Communications of the ACM, 36 No.7, pp. 74–83, July 1993. In Special Issue on Computer-Augmented Environments.
- [10] Bellavista, P., Corradi, A., Stefanelli, C. The ubiquitous provisioning of Internet services to portable devices. IEEE Pervasive Computing, Volume: 1 Issue: 3, pp: 81 –87. July-Sept. 2002.
- [11] Abowd G.D. Software Engineering Issues for Ubiquitous Computing. Proceedings of the International Conference on Software Engineering, ICSE'99, pp. 75-84. 1999
- [12] Abowd G.D. and Mynatt E.D., "Charting Past, Present, and Future Research in Ubiquitous Computing," ACM Trans. Computer-Human Interaction, vol. 7, no. 1, Mar. 2000, pp. 29-58.
- [13] Satyanarayanan M., "Pervasive Computing: Vision and Challenges." IEEE Personal Communications, August 2001.
- [14] Ark W.S., Selker T. A look at human interaction with pervasive computers. IBM Systems Journal, Vol 38, No. 4, July 1999.
- [15] Hansmann U., Merk L., Nicklous M.S., Stober T. Pervasive Computing Handbook. Springer, 2001, pp 341-346.
- [16] Birnbaum J. Pervasive information systems. Communications of the ACM, volume 40, number 2, pp. 40-41. 1997.
- [17] Rhodes B. J., Minar N., Weaver J. Wearable Computing Meets Ubiquitous Computing: Reaping the best of both worlds. The Proceedings of The Third International Symposium on Wearable Computers (ISWC '99), San Francisco, CA, October 18-19 1999, pp. 141-149.

- [18] Edwards K. and Grinter R. At Home with Ubiquitous Computing: Seven Challenges, Ubiquitous Computing 2001. [Online document]. Available URL: (last time visited May 6th, 2003) <http://www2.parc.com/csl/members/grinter/ubicomp.pdf>
- [19] Stanford V. The NIST Smart Flow System: Integrating Intelligent Systems. NIST Pervasive Computing 2001 Conference. May 2001. [Online document]. Available URL: (last time visited May 6th, 2003) (last time updated June 29th, 2001) <http://www.nist.gov/pc2001/agenda.html>
- [20] Rysavy P. The evolution of Cellular Data. 1999. [Online document]. Available URL: (last time visited January 6th, 2003) <http://www.MDI-ng.org/es53061/3G.htm>
- [21] Forman D. An IT Buyer's Guide to Industrial Strength Wide-Area Wireless. Nov. 2000. [Online document]. Available URL: (last time visited May 6th, 2003) (last time updated Nov. 27th, 2000) http://searchnetworking.techtarget.com/originalContent/0,289142,sid7_gci801341,00.htm
- [22] IEEE 802.11b Wireless LANs. 3Com Technical Paper. [Online document]. Available URL: (last time visited May 6th, 2003) http://www.3com.com/corpinfo/en_US/technology/index.jsp
- [23] IrDA (Infrared Data Association). [Online document]. Available URL: (last time visited May 6th, 2003) <http://www.irda.org>
- [24] The Official Bluetooth website. [Online document]. Available URL: (last time visited May 6th, 2003) <http://www.bluetooth.com>
- [25] Sun Microsystems. Java 2 Platform Micro-Edition (J2ME) Technology for Creating Mobile Devices. White paper. May 2000. [Online document]. Available URL: (last time visited May 6th, 2003) <http://java.sun.com/products/cldc/wp/KVMwp.pdf>

- [26] Helal S. Pervasive Java. Pervasive Computing, Vol.1, Issue 2, pp. 85 –89, Apr-Jun 2002.
- [27] IBM VisualAge Micro Edition (VAME). [Online document]. Available URL: (last time visited May 6th, 2003) <http://www.embedded.oti.com>
- [28] VisualAge Micro Edition. Handheld and Wireless Solutions Journal. Issue 2, 2002. [Online document]. Available URL: (last time visited May 6th, 2003) <http://www.wirelessolutionsjournal.com>
- [29] Daniele M., Wijnen B., Ellison M., Francisco D. Agent Extensibility (AgentX) Protocol. RFC 2741. January 2000. Available URL: <http://www.ietf.org/rfc/rfc2741.txt>
- [30] White M., Gudur S. An Overview of the AgentX Protocol. The Simple Times, Volume 6, Number 1, March 1998.
- [31] Strauß F., Schönwälder J., Mertens S. JAX – A Java AgentX subagent Toolkit. Informatik Bericht 2000, July 2000. [Online document]. Available URL: (last time visited May 6th, 2003) <http://www.ibr.cs.tu-bs.de/users/strauss/>
- [32] NET-SNMP Project. [Online document]. Available URL: (last time visited May 6th, 2003) <http://www.net-snmp.com>
- [33] Natale B. AgentX Implementation Report Summary. June, 2001. [Online document]. Available URL: (last time visited May 6th, 2003) <http://www.ietf.org/IESG/Implementations/AGENTX-IMPLEMENTATION>
- [34] J.AgentX. University of Coimbra. [Online document]. Available URL: (last time visited May 6th, 2003) <http://eden.dei.uc.pt/agentx/>

- [35] J. Schonwalder and F. Strauß. Next Generation Structure of Management Information for the Internet. In Proc. 10th IFIP/IEEE Workshop on Distributed Systems: Operations and Management, pages 93–106. Springer Verlag, October 1999.
- [36] Westhawk SNMP stack. [Online document]. Available URL: (last time visited May 6th, 2003) <http://www.westhawk.co.uk/resources/snmp.html>
- [37] Java 2 Platform Micro Edition (J2ME) Wireless Toolkit. [Online document]. Available URL: (last time visited May 6th, 2003) <http://java.sun.com/products/j2mewtoolkit/>
- [38] Message Processing and Dispatching for the Simple Network Management Protocol (SNMP). RFC 3412. [Online document]. Available URL: (last time visited May 6th, 2003) <ftp://ftp.isi.edu/in-notes/rfc3412.txt>

Appendix A

Modifications to the NET-SNMP library

In this appendix the modifications to the NET-SNMP library are presented. The code that has been added is in bold.

The code of the C implementation file “snmpd.c” is:

```
/*
 * snmpd.c
 */
/** @defgroup agent The snmp agent
 * The snmp agent responds to SNMP queries from management stations
 */

#include <net-snmp/net-snmp-config.h>
...
#include <net-snmp/net-snmp-includes.h>
#include <net-snmp/agent/net-snmp-agent-includes.h>

static void
usage(char *prog)
{
    printf("\nUsage: %s [OPTIONS] [LISTENING ADDRESSES]", prog);
    ...

    printf(" -R to support RAgentX \n"); /*Added to support RAgentX */
    ...
    printf(" -x ADDRESS use ADDRESS as AgentX address\n");
    ...
    printf("\n");
    exit(1);
}

...

/*****
 * main function
 * Setup and start the agent daemon.
 *****/

int
```

```

main(int argc, char *argv[])
{
    char    options[128] = "aAc:CdD::fhHl:l:LP:qrRsvV-: "; /*Added -R option*/
    int     arg, i, ret;
    int     dont_fork = 0;
    int     dont_zero_log = 0;
    ...

    while ((arg = getopt(argc, argv, options)) != EOF) {
        switch (arg) {
            case '-':
                if (strcasecmp(optarg, "help") == 0) {
                    usage(argv[0]);
                }
                ...

            case 'R': /*Added to support RAgentX */
                netsnmp_ds_set_boolean(NETSNMP_DS_APPLICATION_ID,
                                      NETSNMP_DS_AGENT_RAGENTX_MASTER, 1);
                break;

            ...

            default:
                usage(argv[0]);
                break;
        }
    }

    init_agent("snmpd"); /* do what we need to do first. */
    init_mib_modules();

    /*
     * start library
     */
    init_snmp("snmpd");

    if ((ret = init_master_agent()) != 0) {
        /*
         * Some error opening one of the specified agent transports.
         */
        Exit(1); /* Exit logs exit val for us */
    }

    ...

    /*
     * Forever monitor the dest_port for incoming PDUs.
     */
    receive();
    /* The "receive" routine has an infinite while-loop that monitors incoming messages for the agent.
     * It invokes the established message handlers for incoming messages on a per port basis. It also
     * handles timeouts.*/

    ...

```

```

    return 0;
} /* End main() -- snmpd */

```

The C code of the function “handle_master_agentx_packet” of the C implementation file “master_admin.c” is:

```

int handle_master_agentx_packet(int operation, netsnmp_session * session,
                                int reqid, netsnmp_pdu *pdu, void *magic)
{
    netsnmp_agent_session *asp;
    struct timeval now;

    if (operation == NETSNMP_CALLBACK_OP_DISCONNECT) {
        /*
         * Shut this session down gracefully.
         */
        close_agentx_session(session, -1);
        return 1;
    } else if (operation != NETSNMP_CALLBACK_OP_RECEIVED_MESSAGE) {
        return 1;
    }

    if (magic) {
        asp = (netsnmp_agent_session *) magic;
    } else {
        asp = init_agent_snmp_session(session, pdu);
    }

    switch (pdu->command) {
    case AGENTX_MSG_OPEN:
        asp->pdu->sessid = open_agentx_session(session, pdu);
        if (asp->pdu->sessid == -1)
            asp->status = session->s_snmp_errno;
        break;

    case AGENTX_MSG_CLOSE:
        asp->status = close_agentx_session(session, pdu->sessid);
        break;

    case AGENTX_MSG_REGISTER:
        asp->status = register_agentx_list(session, pdu);
        break;

    case AGENTX_MSG_UNREGISTER:
        asp->status = unregister_agentx_list(session, pdu);

```

```

break;

case AGENTX_MSG_INDEX_ALLOCATE:
    asp->status = allocate_idx_list(session, asp->pdu);
    if (asp->status != AGENTX_ERR_NOERROR) {
        snmp_free_pdu(asp->pdu);
        asp->pdu = snmp_clone_pdu(pdu);
    }
    break;

case AGENTX_MSG_INDEX_DEALLOCATE:
    asp->status = release_idx_list(session, pdu);
    break;

case AGENTX_MSG_ADD_AGENT_CAPS:
    asp->status = add_agent_caps_list(session, pdu);
    break;

case AGENTX_MSG_REMOVE_AGENT_CAPS:
    asp->status = remove_agent_caps_list(session, pdu);
    break;

case AGENTX_MSG_NOTIFY:
    asp->status = agentx_notify(session, pdu);
    break;

case AGENTX_MSG_PING:
    asp->status = agentx_ping_response(session, pdu);
    break;

case AGENTX_MSG_GET:
    if (netsnmp_ds_get_boolean(NETSNMP_DS_APPLICATION_ID,
                              NETSNMP_DS_AGENT_RAGENTX_MASTER) == 1){
        if (!handle_agentx_packet(operation, session, reqid, pdu, magic))
            printf("Problem with the 'handle_agentx_packet'\n");
        return 1;
    }

case AGENTX_MSG_TESTSET:
    if (netsnmp_ds_get_boolean(NETSNMP_DS_APPLICATION_ID,
                              NETSNMP_DS_AGENT_RAGENTX_MASTER) == 1){
        if (!handle_agentx_packet(operation, session, reqid, pdu, magic))
            printf("Problem with the 'handle_agentx_packet'\n");
        return 1;
    }

case AGENTX_MSG_GETNEXT:
    break;

case AGENTX_MSG_GETBULK:
    break;

case AGENTX_MSG_COMMITSET:
    break;

```

```

case AGENTX_MSG_UNDOSET:
    break;

case AGENTX_MSG_CLEANUPSET:
    break;

case AGENTX_MSG_RESPONSE:
    break;

default:
    asp->status = AGENTX_ERR_PARSE_FAILED;
    break;
}

gettimeofday(&now, NULL);
asp->pdu->time = calculate_time_diff(&now, &starttime);
asp->pdu->command = AGENTX_MSG_RESPONSE;
asp->pdu->errstat = asp->status;
if (!snmp_send(asp->session, asp->pdu)) {
    char      *eb = NULL;
    int       pe, pse;
    snmp_error(asp->session, &pe, &pse, &eb);
    snmp_free_pdu(asp->pdu);
    free(eb);
}
asp->pdu = NULL;
free_agent_snmp_session(asp);

return 1;
}

```

The C code of the function “handle_agentx_packet” of the C implementation file “subagent.c” is:

```

int handle_agentx_packet(int operation, netsnmp_session * session, int reqid, netsnmp_pdu
*pdu, void *magic)
{
    struct agent_netsnmp_set_info *asi = NULL;
    snmp_callback mycallback;
    netsnmp_pdu *internal_pdu = NULL;
    void *retmagic = NULL;
    ns_subagent_magic *smagic = NULL;

    if (operation == NETSNMP_CALLBACK_OP_DISCONNECT) {
        int period = netsnmp_ds_get_int(NETSNMP_DS_APPLICATION_ID,
        NETSNMP_DS_AGENT_AGENTX_PING_INTERVAL);

        /*

```



```

    * Deregister the ping alarm, if any, and invalidate all other
    * references to this session.
    */
    if (session->securityModel != SNMP_DEFAULT_SECMODEL) {
        snmp_alarm_unregister(session->securityModel);
    }
    snmp_call_callbacks(SNMP_CALLBACK_APPLICATION,
        SNMPD_CALLBACK_INDEX_STOP, (void *) session);
    agentx_unregister_callbacks(session);
    remove_trap_session(session);
    register_mib_detach();
    main_session = NULL;
    if (period != 0) {
        /*
         * Pings are enabled, so periodically attempt to re-establish contact
         * with the master agent. Don't worry about the handle,
         * agentx_reopen_session unregisters itself if it succeeds in talking
         * to the master agent.
         */
        snmp_alarm_register(period, SA_REPEAT, agentx_reopen_session,
            NULL);
    }
    return 0;
} else if (operation != NETSNMP_CALLBACK_OP_RECEIVED_MESSAGE) {
    return 1;
}

/*
 * ok, we have a pdu from the net. Modify as needed
 */

pdu->version = AGENTX_VERSION_1;
pdu->flags |= UCD_MSG_FLAG_ALWAYS_IN_VIEW;

if (pdu->command == AGENTX_MSG_GET
    || pdu->command == AGENTX_MSG_GETNEXT
    || pdu->command == AGENTX_MSG_GETBULK) {

    smagic = (ns_subagent_magic *) calloc(1, sizeof(ns_subagent_magic));
    if (smagic == NULL) {
        return 1;
    }
    smagic->original_command = pdu->command;
    smagic->session = session;
    smagic->ovars = NULL;
    retmagic = (void *) smagic;
}

switch (pdu->command) {
case AGENTX_MSG_GET:
    pdu->command = SNMP_MSG_GET;
    mycallback = handle_subagent_response;
    break;

case AGENTX_MSG_GETNEXT:
    pdu->command = SNMP_MSG_GETNEXT;

```

```

/*
 * We have to save a copy of the original variable list here because
 * if the master agent has requested scoping for some of the varbinds
 * that information is stored there.
 */

smagic->ovars = snmp_clone_varbind(pdu->variables);
mycallback = handle_subagent_response;
break;

case AGENTX_MSG_GETBULK:
    pdu->command = SNMP_MSG_GETBULK;
    smagic->ovars = snmp_clone_varbind(pdu->variables);
    mycallback = handle_subagent_response;
    break;

case AGENTX_MSG_RESPONSE:
    return 1;

case AGENTX_MSG_TESTSET:
    asi = save_set_vars(session, pdu);

    // Using RAgentX
    if (netsnmp_ds_get_boolean(NETSNMP_DS_APPLICATION_ID,
                               NETSNMP_DS_AGENT_RAGENTX_MASTER) == 1){

        asi->mode = pdu->command = SNMP_MSG_SET;
    } else
        // It's a AgentX subagent
        asi->mode = pdu->command = SNMP_MSG_INTERNAL_SET_RESERVE1;

    mycallback = handle_subagent_set_response;
    retmagic = asi;
    break;

case AGENTX_MSG_COMMITSET:
    asi = restore_set_vars(session, pdu);
    asi->mode = pdu->command = SNMP_MSG_INTERNAL_SET_ACTION;
    mycallback = handle_subagent_set_response;
    retmagic = asi;
    break;

case AGENTX_MSG_CLEANUPSET:
    asi = restore_set_vars(session, pdu);
    if (asi->mode == SNMP_MSG_INTERNAL_SET_RESERVE1 ||
        asi->mode == SNMP_MSG_INTERNAL_SET_RESERVE2) {
        asi->mode = pdu->command = SNMP_MSG_INTERNAL_SET_FREE;
    } else {
        asi->mode = pdu->command = SNMP_MSG_INTERNAL_SET_COMMIT;
    }
    mycallback = handle_subagent_set_response;
    retmagic = asi;
    break;

case AGENTX_MSG_UNDOSET:

```

```

asi = restore_set_vars(session, pdu);
asi->mode = pdu->command = SNMP_MSG_INTERNAL_SET_UNDO;
mycallback = handle_subagent_set_response;
retmagic = asi;
break;

default:
    pdu->command, pdu->command));
    return 0;
}

/*
 * submit the pdu to the internal handler
 */

/*
 * We have to clone the PDU here, because when we return from this
 * callback, sess_process_packet will free(pdu), but this call also
 * free()s its argument PDU.
 */

internal_pdu = snmp_clone_pdu(pdu);

snmp_async_send(agentx_callback_sess, internal_pdu, mycallback, retmagic);

return 1;
}

```