Mobile Software Agents for Network Monitoring and Performance Management

Damianos Gavalas

A thesis submitted for the degree of Doctor of Philosophy

Department of Electronic Systems Engineering

University of Essex

June 2001

ABSTRACT

The rapid developments in the area of communication and data networks require high-performance and reliable Network and Systems Management (NSM). Due to the deficiencies of the traditional manager-agent paradigm and the resulting *scalability* and *flexibility* limitations of centralisation, it is now commonly agreed that a transition to *distributed management* is required. There are many new trends in NSM that try to address this requirement. In general, these trends make use of well-established techniques from other computing fields, e.g. distributed objects, Java and Web technologies, adapted to NSM problems. In addition, management distribution aspects have also been incorporated in both the Internet and OSI-SM management architectures. However, it has been shown that distribution alone is not enough. Due to the high frequency of changes in networking environments, it is imperative that management systems are flexible, i.e. they can dynamically adapt to those changes and allow on-the-fly customisation of management services. In that context, *Management by Delegation* (MbD) paradigm has been identified as the first promising approach towards management distribution and flexibility.

Mobile Agents (MA) represent a relatively new computing paradigm used in several domains as a means for structuring distributed applications. The term MA refers to a software entity identified by its code, persistent and execution state, with the ability to autonomously move from host to host, acting on behalf of a user or an application. The MA paradigm is part of the generic family of *mobile code* paradigms, which provide a mechanism to extend the capabilities of services through dynamically moving the components of a distributed application among the nodes of a network. In the context of NSM, MAs represent a relatively new trend that builds on the groundwork of MbD and involves the use of mobile objects as a tool to realise scalable and flexible distributed management systems. MA-based management has been a subject of intense research in the past few years, reflected on the proliferation of MA platforms (MAP) expressly oriented to distributed NSM. However, the applicability of MAs in management applications is still subject of ongoing debates, due to concerns related to the security, performance and scalability aspects of this technology. Namely, it is still unclear which problems of distributed management can be addressed by MAs and whether their use can offer performance benefits compared to alternative solutions.

A first objective of the research work described in this thesis has, therefore, been to assess the values and weaknesses of MA technology from the management viewpoint and identify ways for the effective use of this technology in distributed management applications. Based on this assessment, we have identified a set of requirements that MAPs tailored to management applications should meet. Following that, we have implemented an NSM-oriented MA framework, satisfying these requirements. Our framework addresses the scalability, flexibility and adaptability requirements related to the management of large-scale and dynamic networks. The performance of our prototype has been quantitatively and experimentally evaluated, demonstrating its competence against centralised management and distributed objects-based approaches.

In addition, three novel applications that exploit the ability of MAs for performing decentralised operations have been developed on the top of the introduced MA-based management framework. The application areas include the fields of *network monitoring* and *performance management*. In particular, the proposed applications include semantic compression of management data, efficient retrieval of bulk monitoring data and filtering of SNMP tables according to pre-defined, arbitrarily complex filtering criteria.

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude towards my academic supervisor, Prof. Mohammed Ghanbari, for his guidance and support throughout the course of this work. I also feel indebted to my second supervisor, Prof. Mike O'Mahony, for giving me the opportunity to start my PhD, his inspiring suggestions and useful advice. I am grateful to Dr. Dominic Greenwood, my 'industrial supervisor', for introducing me to the fields of mobile agents and network management, keeping me on the rails (especially in the first stages of my research) and treating me more like a friend rather than a colleague. Fujitsu Telecommunications Europe Ltd. is acknowledged for funding this work.

Also, the members of the UCD-SNMP, Aglets and mobility mailing lists have provided an invaluable service as a community of peers amongst which to discuss my research. Many of the ideas expressed in this thesis have been shaped and refined in those forums. Many thanks to Wes Hardaker, Todd Papaioannou and especially Kristen Pol that helped me to solve a Java programming problem which troubled me for over two months. Paolo Bellavista and Prof. Antonio Corradi are also acknowledged for their valuable tips and insightful ideas during the implementation of the Resources Usage Monitoring Tool described in Chapter 6.

I would also like to thank all my current and past colleagues at the research laboratory who have created an enjoyable atmosphere: Tom Cambel, Jason Neale, Noel Parnis, Alicia Calonico-Soto, Fyllon Papadopoulos, Costas Dimopoulos, Bernard Lee, Jenny He, Candies Lam, Antonio Pinheiro and Moises Ribeiro. It has been a genuine pleasure to work with them. Special thanks to Moises who has been the best officemate I could ever ask for; I have very much enjoyed our long, non-technical conversations.

I would like to acknowledge the help of all my friends: for keeping me 'mentally' healthy and most importantly for reminding me that there are more important things in life than a PhD. Many thanks to Kostas Drakos, George Spiliotopoulos, Alkistis Deligianni, Tasos Fotiou, Maria Ziouvelou, Panos Hanjiprokopiou, Antonis Blahos, Deni Alexandropoulou, Thomas Verakis, Zambia Tzouganaki, Nikos Apostolopoulos, Baggelis Katsaras, Giorgos Parargiris.

Special thanks to my dearest friends Dimitris Alexandropoulos, Tanya Politi and Ottavia Jedrkiewicz; they have all been very caring and supportive and have greatly contributed to making the last and most stressful year of my studies an adventure worthwhile to live.

Most importantly I would like to thank Katerina Proestaki, whose unswerving support and companionship have allowed me to concentrate my efforts on achieving my goals and made this long and difficult trip look like a pleasant journey. Katerina has also reviewed and provided valuable feedback on several of my research papers published over the last three years and also on Chapters 1 and 2 of this thesis.

Finally, I would like to thank my family, Giorgos, Maria, Nikos and Dimitris Gavalas for caring, supporting and always being there for me throughout my many years of study. They have probably contributed much more than me to this thesis. I hope my completion will repay to some extend their trust, support and expectations.

Στην οικογένεια

και τους φίλους μου

“Να υποφέρεις δεινά που ακόμα και η ελπίδα τα θεωρεί ατέρμονα
Να αψηφάς την εξουσία που μοιάζει παντοδύναμη
Να αγαπάς και να υπομένεις
Να ελπίζεις, εώς ότου η ελπίδα μέσα από την ίδια την καταστροφή της, να πραγματώσει το σκοπό της
Να μην αλλάζεις, ούτε να διστάζεις, μήτε να μετανιώνεις˙
εδώ είναι η δόξα σου Τιτάνα: καλοσύνη, μεγαλείο, ευτυχία, ομορφιά και ελευθερία..„

“Προμηθέας λυόμενος”
Α. Σέλλευ

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ACL	Agent Communication Language
ADK	AgentBean Development Kit
AN	Active Network
API	Application Programming Interface
ASN.1	Abstract Syntax Notation 1
ATM	Asynchronous Transfer Mode
ATP	Agent Transfer Protocol
BDK	Bean Development Kit
BER	Basic Encoding Rules
CCITT	Comittee Consultif International Telegraphique et Telephonique
CL	Class Loader
CLD	Class Loader Daemon
CIM	Common Information Model
CMIP	Common Management Information Protocol
CMIS	Common Management Information Service
CMST	Constrained Minimum Spanning Tree
COD	Code On Demand
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CS	Client/Server
DES	Data Encryption Standard
DLL	Dynamic Link Library
DISMAN	DIStributed MANagement
DMI	Desktop Management Interface
DMTF	Distributed Management Task Force
DNS	Domain Name Service
DOT	Distributed Objects Technology
DSA	Digital Signature Algorithm

E-W	Esau-Williams
FIPA	Foundation for Intelligent Physical Agents
FTP	File Transfer Protocol
GIOP	General Inter-ORB Protocol
GnG	Get ’n’ Go
GnS	Go ’n’ Stay
GUI	Graphical User Interface
HF	Health Function
HTML	HyperText Meta-Language
HTTP	HyperText Transfer Protocol
ID	Identification
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
I/O	Input/Output
ISO	International Standards Organization
ITU-T	International Telecommunication Union—Telecommunications standardisation
JAR	Java Archive
JCA	Java Cryptography Architecture
JDBC	Java DataBase Connectivity
JDK	Java Development Kit
JDMK	Java Dynamic Management Kit
JIT	Just-In-Time
JMAPI	Java Management API
JMX	Java Management eXtensions
JNI	Java Native Interface
JRMP	Java Remote Method Protocol
JVM	Java Virtual Machine
LAN	Local Area Network
ICMP	Internet Control Message Protocol
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
ILP	Integer Linear Programming
IP	Internet Protocol
LDAP	Lightweight Directory Access Protocol
M2M	Manager-to-Manager

MA	Mobile Agent
MAC	Medium Access Control
MACL	Mobile Agent Class Loader
MAG	Mobile Agent Generator
MAL	Mobile Agent Listener
MAP	Mobile Agent Platform
MAR	Mobile Agent Register
MAS	Mobile Agent Server
MbD	Management by Delagation
MCR	Mobile Code Repository
MD5	Message Digest
MDM	Mobile Distributed Manager
MFC	Migration Facility Component
MIB	Management Information Base
MLM	Mid-Level Manager
MO	Managed Object
MST	Minimum Spanning Tree
NDD	Network Discovery Daemon
NE	Network Element
NMS	Network Management System
NSM	Network & Systems Management
OIP	Optimal Itineraries Planning
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open Systems Interconnection
PC	Personal Computer
PDU	Protocol Data Unit
PI	Polling Interval
PID	Process Identifier
PT	Polling Thread
PTC	Polling Thread Configuration
QoS	Quality of Service
RDP	Remote Delegation Protocol
REV	REmote Evaluation
RFC	Request For Comments
RI	Resources Inspector

RIA	Resource Inspection Application
RMI	Remote Method Invocation
RMON	Remote MONitoring
RMT	Resources Monitoring Tool
RPC	Remote Procedure Call
RSA	Rivest-Shamir-Adleman
SC	Security Component
SD	Standard Deviation
SFC	Service Facility Component
SHA-1	Secure Hash Algorithm
SLA	Service Level Agreement
SM	Security Manager
SMF	System Management Functions
SNMP	Simple Network Management Protocol
SQL	Standard Query Language
TCP	Transmission Control Protocol
TF	Table Filterer
TINA	Telecommunications Intelligent Network Architecture
TMN	Telecommunications Management Network
TP	Table Poller
UDP	User Datagram Protocol
URL	Universal Resource Locator
VDL	View Definition Language
WAN	Wide Area Network

CHAPTER 1

INTRODUCTION

1.1. NETWORK AND SYSTEMS MANAGEMENT: THE PATHWAY TO DISTRIBUTED MANAGEMENT

The need for data communications has evolved rapidly since the earliest days of computing. Industrial enterprises are increasingly dependent upon networked systems serving as their information backbones. This dependency increases in turn the requirement for reliable operation of the underlying communication and data infrastructure used within a business environment. Faults and performance inefficiencies in these systems give raise to considerable business losses. Therefore, the *management* of network infrastructures, including communication elements, systems and applications becomes imperative. According to the definition given in [HEG94], the term *Network and Systems Management* (NSM) refers to the "sum of all procedures and products for planning, configuring, controlling, monitoring and managing computer networks and distributed systems and removing errors in these. This should provide user friendly and economic support for users working with the network and its components. Thus, it covers all the precautions and activities needed to ensure the effective and efficient use of the resources to be managed."

The current globalisation of networks and services imposes an inevitable need for decentralisation of computing processes. The transition from centralised mainframe systems to distributed systems is another contributing factor. Despite the benefits of the distributed approach, new problems arise in the management of such systems. The constantly increasing heterogeneity of network components, communications protocols, end systems and applications also increases the complexity of the management system.

NSM should effectively address these problems. It is commonly accepted that the management of large-scale networking environments cannot be adequately addressed through centralised approaches that characterise contemporary management systems [GOL91].

Concentration of functionality, intelligence and responsibilities at a central site represents a rigid management solution that suffers from the well-known problems of centralised systems. Existing management servers (*agents*) transfer unfiltered and unprocessed management information to the central *management station* [GOL98]. This process incurs high network traffic for NSM purposes and leads to network management systems (NMS) with poor *scalability*. As a consequence, decentralised, distributed solutions are required to address the limitations of centralised archetypes. This requirement has been recognised by researchers since the early 90s, with *distributed management* becoming a subject of intense research [KAH97, MAR99a]. However, the Simple Network Management Protocol (SNMP) [CAS90], which is the dominant protocol in the Internet management, defines a centralised architecture.

Management distribution requirements are partially addressed by hierarchical NMSs, which enable data pre-processing and delivery of higher-level information to the manager station [KOO95, SIE96, MLM]. In such models, parts of the management functionality is permanently assigned to statically defined managing entities (*mid-level managers* or *proxies*) running on specific nodes within the distributed environment. Such functionality should be assigned at the managing entities design phase and cannot be easily modified [MOU98a]. Hence, it cannot cope well with the high dynamics of modern networks, referring to frequent changes that need to be incorporated in the NMSs as well. Those changes may vary from the introduction of new devices and software to the provision of new types of services. The frequency of such changes reflects in the amount of Requests for Comments (RFC) produced every year in the Internet management domain [RFCs].

Management distribution aspects are also addressed in the OSI Systems Management (OSI-SM) architecture, allowing to delegate monitoring activities to network elements (NE), reporting only important events or summarised reports to higher-level managers [ISO92, ISO93]. Similar facilities have been later incorporated in SNMP environments [WAD95], whilst recent approaches proposed the provision of such facilities through distributed object technologies [MAZ96, PAV00], exemplified by the Common Object Request Broker Architecture (CORBA) [CORBA]. Yet, these approaches fail to meet management flexibility and reconfigurability requirements due to the rigid and static distribution of functionality that can be only modified through a complex and time-consuming research, standardisation, implementation and deployment cycle [BOH00b].

The requirement for *flexibility* combined with that for *decentralisation* has been addressed by the pioneering work of Goldszmidt et al. who introduced the concept of *Management by Delegation* (MbD) [GOL91]. Delegation of management functionality refers to moving part of the intelligence traditionally residing on a central management station to distributed management entities running on selected managed systems. This allows the dynamic change of responsibilities and tasks allocation and leads to flexible systems with dynamically augmented capabilities. A main objective of MbD is to perform decentralised tasks in proximity to system resources to prevent unnecessary data transfers, while easing the introduction of new tasks at runtime. The ideas behind MbD have inspired many proposals [KAL97, MOU98a], while subsequent research demonstrated its applicability both in the context of OSI-SM [VAS97] and SNMP [LEV99].

MbD can be considered a precursor of the ideas discussed in this thesis. Nevertheless, approaches based on the *code mobility* paradigm cover broader aspects of management delegation and can include MbD as a special case. Code mobility refers to the capability of dynamically moving the components of a distributed application among the nodes of a network [CAR97]. This idea can be applied in the generic field of distributed systems, however distributed management represents a very promising application domain, as it can benefit form the efficient use of bandwidth resources and the high degree of flexibility and reconfigurability offered by mobile code [BAL97].

A sub-class of the code mobility paradigms, termed the *Mobile Agent* (MA) paradigm, has emerged in the mid-90's. Briefly, MAs can be defined as *computational components identified by their code, persistent and execution state, with the ability to autonomously move from host to host, acting on behalf of a user or an application* [PHA98]. In the distributed management domain, MAs add a new dimension by exploiting their ability to autonomously migrate to different network devices and perform complex management tasks without the manager's intervention [PIC98]. The *migration* and *autonomy* features of MAs attracted the attention of many researchers and, as result, MA-based distributed management has become a hot research topic in the past few years [BIE98b, CHE98, MAR99a].

1.2. Mobile Agents for Network Management: Prospects & Hurdles

Since its inception, MA technology soon became very popular in distributed computing field. Several research activities investigated its applicability in numerous application domains, including information retrieval [ISM99, PAP99], e-commerce [DAS99], mobile computing [CHE95, MIL99] and parallel computation [SIL99b, GHA99]. Yet, MA technology has not yet gained large acceptance by the developers of real applications, despite the large number of interesting experiences already available. This is mainly due to several technical and non-technical hurdles that must be addressed by MA community to make possible the wide spread of this technology: the lack of killer applications [MIL99, KOT99], security [NWA96] and performance [PIC01].

The advent of MA technology has caused a serious impact on distributed management research. Especially at the time this ‘exotic’ technology was still at its infancy, it was considered by many as a particularly promising vehicle that could lead management research towards the ultimate goal of fully distributed and automated management [MAG96, BIE97]. As a result, numerous research projects have been established aiming at investigating the potential of MA technology in the management domain. Some of them represent academic initiatives [AMETAS, MAP, SOMA], others comprise collaborative activities among academic and commercial partners [JAMES], whilst many have been funded by the Commission of the European Union [MIAMI].

As a result of the intense research on MA technology, we have observed the proliferation of available mobile agent platforms (MAP) over the last few years, currently reaching the impressive number of more than seventy known platforms [MAL], a lot more than available RPC or CORBA implementations [MAR01]. Many of these platforms are expressly oriented to management applications (e.g. [SAH97, SUS98, NIC98, SIL99a, PUL00a]), aiming at optimising flexibility and performance aspects not sufficiently addressed by general-purpose platforms [Voyager, Grasshopper]. These MAPs have been used as the basis for covering a wide spectrum of management applications, including network routing [APP94], network monitoring [KU97, ZAP99, PIN99], fault [ELD99], configuration [PAG98], performance [BOH00a], security [LAZ97] and service management [KNI99] areas. These applications demonstrated the applicability of MAs on several management scenarios, exploiting their migration and autonomy features to dynamically customise the management capabilities of network devices, perform semantic compression of data, or detect changes related to performance, faults or configuration of managed systems.

Although these approaches are in the right direction, there are several limitations related to their practical use on distributed management. These limitations relate to both the design of MAPs tailored to management applications and on the applications themselves. In particular, existing MAPs use *heavyweight protocols* to enable agent mobility, where in most of the cases both the agent’s code and state are transferred on every MA transfer. That scheme introduces unnecessary network overhead as the code transfer can be omitted in cases that the MA has visited a network device in the past. In addition, management MAPs lack mechanisms to automate the effortless *introduction of MA-based management tasks* and also to dynamically *modify the functionality of distributed management services*, i.e. to update MAs code. Finally, these MAPs define simplistic, two-level MA *organisation models*, which are not suitable for the management of large-scale, geographically dispersed enterprise networks [LIO01] and does not allow the *dynamic adaptation* of the management framework to the changing topology and traffic characteristics of such networks. These issues are addressed in the design of a lightweight, flexible, scalable and adaptive MA-based management infrastructure described in Chapters 4-6.

Referring to MA-based management applications, most of the approaches *concentrate on the mobility aspect* of the paradigm, enforcing the use of 'multi-hop' MAs (e.g. [KU97, SAH98, PUL00b]). Although this mobility scheme can be useful in some cases, it is not always the most appropriate management delegation technique [PIC01]. In fact, when the mobility feature is not exploited to enable semantic compression and correlation of management data, this approach can prove less scalable than traditional centralised solutions [BAL98, LOP00]. On the other hand, there is a range of applications that can be more efficiently implemented using 'single-hop' agents [BOH00c, MAR01]. We argue that the selection of the most suitable mobility scheme should be application-dependent. This issue is investigated throughout this thesis and applications that can benefit either by single-hop or multi-hop agents are identified.

1.3. THESIS WORK

At the time our research commenced (October 1997), the field of MA-based management was still unexplored. Research papers describing the implementation experiences gained from the first prototypes were published in that period [SAH97, KU97, BIE97]. Hence, the assets and limitations related to the practical use of MAs in the management domain were still unclear.

A first objective of the research work described in this thesis has, therefore, been to assess the values and weaknesses of MA technology from the management viewpoint. Based on this assessment, we have identified a set of requirements that MAPs tailored to management applications should meet. Following that, we have implemented an NSM-oriented MA framework, satisfying these requirements. The design and implementation details of this MAP are given in Chapters 4-6. In particular, Chapter 4 details the implementation of a lightweight MAP with core functionality that enables the effortless development of management applications. Chapter 5 describes extensions on that core framework, addressing the scalability problems raised when used for the management of large sets of NEs, whilst Chapter 6 introduces further optimisations, improving the framework's performance and flexibility when considering the management of hierarchical, large-scale enterprise networks. It is emphasised that throughout these chapters, we derive mathematical formulations that model the performance of our framework and confirm its competence against alternative approaches. The conclusions deduced from these quantitative evaluations are also empirically verified through experimental results.

The management application domains that we have chosen as case studies include two strongly related areas: *network monitoring* and *performance management*. The former refers to monitoring network devices health to ensure that they stay within acceptable parameters [JIA00], whilst the later supports the gathering of statistical data, upon which it applies various analysis routines to measure systems performance [SLO94]. Modern NMSs carry out monitoring and performance management tasks following a data-intensive centralised approach whose limitations have been discussed in Section 1.1. Hence, the flexibility and scalability aspects of these two complementary fields can benefit from MA-based approaches. In Chapter 7, we introduce three novel applications that exploit the ability of MAs for performing decentralised operations to apply semantic compression of management data, efficiently retrieve bulk monitoring data and filter the contents of SNMP tables according to pre-defined, arbitrarily complex filtering criteria.

The emphasis of this work is given on both theoretical and practical aspects of MA-based management. It is evident that the employment of a new computing paradigm in a particular application domain needs first to be theoretically approached and also qualitatively and quantitatively evaluated so as to identify ways to be effectively utilised. However, a theoretical study has no value if not supplemented by prototypical implementations, which provide hands-on experience and allow revealing problems related to the use of this paradigm on real-world applications.

Summarising, the purpose of this thesis is to *investigate* the use of MAs on management applications. Although distributed management represents a particularly promising domain for the use of MA technology, it is now clear that MAs are not always the ideal solution for structuring management applications. In this thesis, we examine the merits and limitations of MA-based management through following an unbiased approach, that aims at identifying scenarios where agent mobility can *really* benefit to build more flexible, scalable and robust management systems.

1.4. Overview and Style of the Thesis

This thesis is organised in the following fashion. This chapter was the introduction to the thesis. Chapters 2 and 3 present state-of-the-art work (literature review) on NSM and MA-based management respectively. Chapters 4, 5, 6 and 7 constitute the core material of the thesis, each addressing a set of related topics. Finally, Chapter 8 summarises the thesis, presents the conclusions and gives directions for future work.

Chapter 2 comprises two main parts. The first part introduces the reader to the main architectural aspects of OSI-SM and Internet management, pointing out their similarities and differences. The architecture, evolution and limitations of SNMP are discussed in detail and emerging technologies in Internet management are reviewed. The second part focuses on distributed management, and presents a state-of-the-art report on standardisation and research initiatives on that field.

Chapter 3 also comprises two main parts. The first part provides an introduction to code mobility paradigms, with particular emphasis on MA technology, and explains their relevance to NSM. Aspects related to agent mobility and MAPs design, along with the performance issues related to their practical use, are also discussed. The focus of the second part is on the application of MA technology on network management. A state-of-the-art of research activity on the field is presented and ways to effectively utilise MAs in management applications are identified.

Chapter 4 provides a detailed analysis of our MA framework design and implementation issues. The main goals of the framework's design are identified, the functionality of its main building components is described in detail and mathematical formulations are derived to evaluate its performance. The performance of the proposed framework is also assessed through a series of experiments, measuring the latency and network overhead incurred when used for management operations.

Chapter 5 describes extensions to the core framework presented in Chapter 4, aiming at maximising its performance and addressing scalability problems raised when the management of large sets of network devices is considered. In particular, it introduces two complementary polling schemes where the first is tailored to obtaining real-time management data and aims at minimising latency, while the second is intended for the collection of data to be analysed off-line and aims at minimising network overhead. The performance gain of the two proposed schemes over standard MA-based polling is verified by a quantitative evaluation and experimental results.

Chapter 6 describes further optimisations to our framework that improves its performance and flexibility aspects when considering the management of hierarchical, geographically dispersed networks. In particular, it introduces a highly scalable and flexible MA-based hierarchical management model with the ability to dynamically adapt to the evolving topological and traffic characteristics of large-scale enterprise networks. The scalability of the proposed framework is, again, verified by a quantitative evaluation and empirical results.

Chapter 7 describes three novel applications on network monitoring and performance management, which address the deficiencies of SNMP when considering bulk management data retrievals. Special focus is given on efficiently retrieving and filtering SNMP tables. The applications have been developed on the top of the MA framework introduced in the previous three chapters. The results of a number of experiments evaluating the performance of the proposed applications are also presented.

Chapter 8 brings together the various strands of the thesis and summarises the main findings. It explains the significance of this thesis and also identifies areas in which the work presented herein could be developed further. Bibliographical references follow at the end, together with a number of appendices.

Considering the style of this thesis, a number of Java object class specifications and code extracts are presented in Chapters 4, 6 and 7 in order to demonstrate the various features of the proposed software infrastructure. These are fundamental as they present the relevant properties and the functionality of the framework's main building blocks, so they could have *not* been moved to appendices. In addition, the presentation of empirical/experimental results includes in several cases 'redundant' information, as the results are communicated in both graphical and tabular form. We have again chosen not to put the tables to appendices as they are strongly correlated with their corresponding graphical representations, highlight the differences between curves that cannot be observed with bare eyes and finally provide complementary information, i.e. additional measurements, standard deviations, etc. Finally, as the reader may have already noticed, "we" or "our" is used when describing research work by the author throughout this thesis.

CHAPTER 2

NETWORK & SYSTEMS MANAGEMENT: STANDARDS, EMERGING TECHNOLOGIES AND THE SHIFT FROM CENTRALISED TO DISTRIBUTED PARADIGM

2.1. INTRODUCTION

In recent years, the use of information resources has dramatically increased, with organisations becoming more and more dependent upon reliable access to this information in order to remain competitive. The explosion in the size and complexity of today's local and wide area networks, combined with the increasing demands placed upon them for their resources resulted in establishing Network & Systems Management (NSM) as a factor of vital importance. To further complicate matters, the concept of single-vendor networks has vanished long time ago. Modern enterprise networks exhibit remarkable diversity of network and systems equipment necessitating the transition to *integrated management*. The primary objective of NSM is to maintain network and systems availability and health, aid in configuring the network and systems, guarantee Quality of Service (QoS), enhance performance, provide security, minimise operational overhead (execution of repetitive tasks) and decrease the cost of running the information technology infrastructure [HEG94].

As a consequence of sharing and interconnecting resources, NSM needs to meet the challenges of distribution, heterogeneity and transparency. A number of approaches and architectures that aim at standardising the management process and addressing the heterogeneity problem have, therefore, emerged. These approaches specify *management architectures*, which supply frameworks for standards of relevance to NSM. Two of the most widely used architectures are the Internet management architecture [HAR99] and the Open Systems Interconnection Systems Management (OSI-SM) architecture [ISO91a]. Other architectures, mainly applied on the telecommunication networks area include the Telecommunications Management Network (TMN) [CCITT92] and the Telecommunications

Intelligent Network Architecture (TINA) [TINA93]. More recent efforts focus on the definition of emerging management architectures, such as Web-Based Enterprise Management (WBEM) [WBEM], Java-based management [JBM] and Directory Enabled Networks (DEN) [STR99]. Distributed objects technologies, exemplified by the Common Object Request Broker Architecture (CORBA) [CORBA], represent another interesting approach which gains increasing attention in the management world.

The Internet management architecture has been criticised for exhibiting low degree of scalability[1], flexibility and reconfigurability, mainly attributed to its *centralised* architecture. The latter two weaknesses also characterise the OSI-SM framework. The emerging management technologies have only partially addressed these problems. The need for *distribution* of NSM functionality has been early recognised by researchers and developers active in this area and several initiatives have been undertaken in this direction; in fact some of them have already led to the specification of standards. However, there is still a long way to go before NSM distribution-related problems are satisfactorily addressed.

The remainder of this chapter starts with a description of the management functional areas identified by the ISO, in Section 2.2. An overview of OSI-SM and Internet management follows in Sections 2.3 and 2.4 respectively, highlighting the similarities and the differences of the two approaches. Special focus is then given to Internet management, describing the architecture, versioning history and limitations of its de-facto standard, the Simple Network Management Protocol (SNMP). Following that, several emerging technologies in the NSM field are described, including distributed object technologies (Section 2.5), Directory Enabled Networks (Section 2.6), Java-based management (Section 2.7) and WBEM (Section 2.8). The problems of centralised management are detailed in Section 2.9, while Sections 2.10 and 2.11 overview the most well-known approaches in distributed management, which however, address only certain aspects of the entire problem area; standardisation and research approaches are separately investigated. Section 2.12 comprises a synthesis and qualitative analysis of distribution approaches, classifies them in *static* and *dynamic* (depending on the degree of flexibility they offer) and identifies their strong and weak points, pointing out aspects of distributed management that could benefit by Mobile Agent (MA)-based approaches. Finally, Section 2.13 summarises the chapter.

It is emphasised that that the reader is assumed to be reasonably familiar with NSM concepts, SNMP, the way SNMP-based management platforms work in the IP world, how they

[1] *Scalability* is defined as the ability to increase the size of the problem domain with a small or negligible increase in the solution’s time and space complexity [RES97]. In the NSM context, scalability is specifically defined as the ability to increase the number of monitored entities or the polling frequency, with a small or negligible decrease in performance.

are typically structured, and the management tasks they perform (for an introduction, see Rose [ROS96] and Stallings [STA99]). We therefore do not redefine here well-established concepts, but only those whose definition is not consensual and those that are relevant to this work.

2.2. MANAGEMENT FUNCTIONAL AREAS

Types of management activity have been categorised by ISO into five generic functional areas, collectively known as FCAPS from their initials [ISO91a]:

- *Fault Management*: the process of collecting information referring to network elements (NE) health. Integrated fault management systems receive reports about malfunctions (alarms), perform alarm correlation and diagnostic tests, identify faults and display various network alarms. This process can be optimised to perform root-cause analysis and suggest/take corrective measures.
- *Configuration Management*: controls the configuration state of a system/network and the relationships between components. It also initialises, configures and shuts down network equipment.
- *Accounting Management*: defines how network usage, charges and costs are to be identified in the networking environment. It is associated with tariffing schemes that generate charging/billing information.
- *Performance Management*: supports the gathering of statistical data, upon which it applies various analysis routines to measure the system performance. That way, it provides an accurate picture of network components and services. This process is capable of proactively pinpointing and forecasting potential problems before they actually occur, based on gathered information. It can predict congestion/bottlenecks and, hence, be used for network future expansions and capacity planning. Performance management represents a central application area for this thesis.
- *Security Management*: controls access to network, system, service and management components. It can offer authentication, confidentiality, integrity, access control and also handle cryptographic key distribution.

In the telecoms world, management platforms generally support most (if not all) of the OSI functional areas. This is not the case in the IP world, where most platforms support only a fraction of FCAPS. Indeed, management platforms are often simpler in the IP world than their counterparts in the telecom world.

2.3. THE OSI SYSTEMS MANAGEMENT

The OSI-SM [ISO91a] defines a management architecture with well-defined *organisational*, *informational*, *communication* and *functional* models. The organisational model assigns special roles to the management entities: the *manager* and the *agent*. A manager is an entity that controls the management process and makes decisions based on collected information, whereas the agents make available the management information to managers. Abstractions of system/network resources which need to be managed are represented by *managed objects* (MO). MOs encapsulate the underlying real resources and enable their manipulation through well-defined operations. An agent administers the MOs on its local device and provides mechanisms for performing management operations upon them, offering an interface to system resources. In essence, the agent acts as *name server* for the objects (resolves their names to internal handles), *object factory* since it creates and maintains objects and *event server* since it disseminates events (notifications are evaluated and forwarded as events to managers according to criteria preset by them) [PAV01]. The communication between the manager and the agents takes place via standardised management protocols. In general, the manager-agent paradigm can be thought of as client/server (CS) relationship, where the manager plays the role of the client and the agent the role of the server.

OSI-SM is based on a complex, object-oriented information model. MO classes are specified by templates and consist of attributes, operations that can be applied to the corresponding objects, behaviours exhibited by the objects in response to operations, and notifications that can be emitted by the objects. The functionality of a MO is defined at design time, i.e., it cannot change at runtime. MOs are logically grouped in *Management Information Bases* (MIB). MIBs are virtual, hierarchical, object-oriented databases including interrelated MOs in a managed environment. The basic architecture of OS-SM is depicted in Figure 2.1.

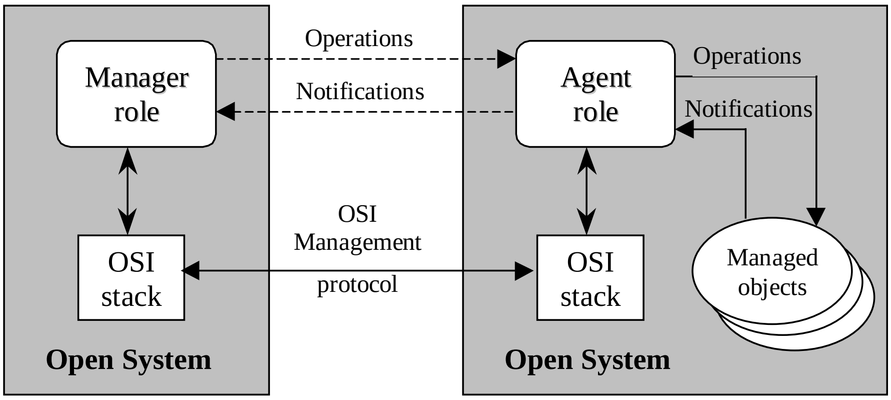


Figure 2.1. Basic OSI-SM architecture [SLO94]

The exchange of management information between managers and agents is defined by a service, the Common Management Information Service (CMIS), and its protocol, the Common

Management Information Protocol (CMIP) [ISO91b]. The CMIS provides management operation primitives that include M-GET to retrieve data, M-SET to modify data, M-ACTION to request the execution of an action, M-CREATE (M-DELETE) to request the creation (deletion) of an instance of a managed object, and M-CANCEL-GET to cancel an outstanding M-GET request. Agents may report events about managed objects using M-EVENT-REPORT. In addition, CMIS provides multiple-object access through *scoping* and *filtering* operations. Scoping allows management operations to be carried out on a selection of one or more managed objects. Filtering consists of boolean expressions with assertions on values of attributes in an object [ISO91b].

OSI-SM also addresses some aspects of management decentralisation through the standardised Systems Management Functions (SMF), which define a rich set of functionality specified in terms of generic object classes. Examples include *Metric Objects*, which measure resource performance, monitor thresholds and generate notifications [ISO93] and the *Summarization* function [ISO92], which provides a framework for the definition, generation, and scheduling of system information summary reports. These functions move intelligence in proximity to the managed resources, reducing the amount of management traffic and providing support for a sophisticated event-driven operation paradigm. Pavlou et al. proposed additional functionality that combines the capabilities of metric monitoring and summarization objects in a powerful fashion [PAV96].

The widespread interest in formal standards has generated considerable interest in CMIP, even though it has not been widely used. One factor contributing to the lack of CMIP's popularity is the slow evolutionary process of these standards.

2.4. INTERNET MANAGEMENT

Simplicity and small implementation overhead has always been the main objective of Internet management since the early days of its conception [CAS90, MCG91]. This seems to be the main reason that justifies its popularity and wide use. In this context, Internet management is characterised by a simple information model that lacks the object-orientation (MOs are nothing more than simple variables) and sophistication of its OSI-SM counterpart, but enables easier and faster writing and instrumentation of MIBs. The architecture depicted in Figure 2.1 is applied in the Internet management model as well; the difference is that the functionality of the corresponding modules is very much simplified.

The Internet management communication model relies on the Simple Network Management Protocol (SNMP) [CAS90] as the communication protocol, which defines connectionless services and primitives for getting and setting variable values and sending notifications. SNMP has been developed with an orientation to TCP/IP networks. As it is the case with most protocols of this kind, its development and implementation occurred with considerable speed. A quick, easy and simple implementation was the first priority of its designers. Hence, the following guidelines have been adhered to:

- make it work over very uncomplicated protocols;
- keep the number of protocol message types small;
- stick to a unit of information that is a single value, such as an integer or string.

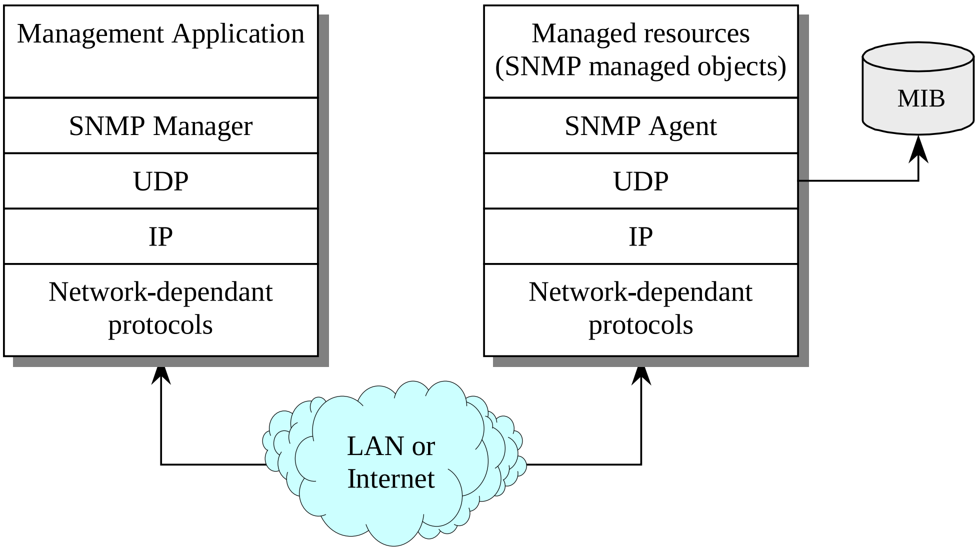

Management Application
SNMP Manager
UDP
IP
Network-dependant protocols
Managed resources (SNMP managed objects)
SNMP Agent
UDP
IP
Network-dependant protocols
MIB
LAN or Internet

Figure 2.2. The SNMP layering [SLO94]

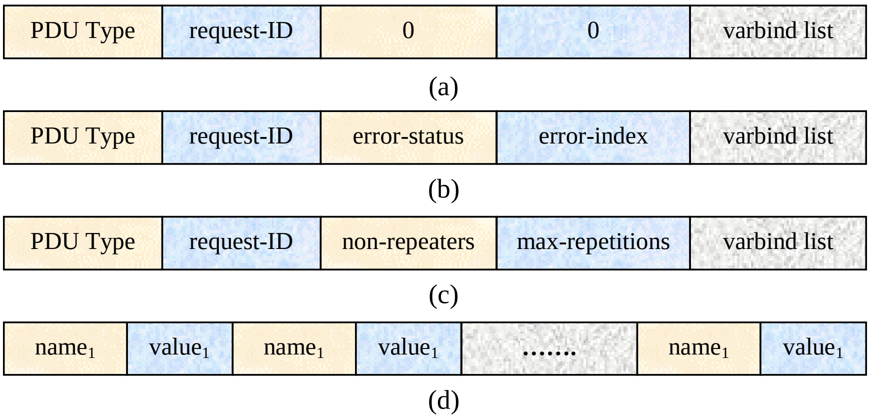
The User Datagram Protocol (UDP) was chosen as the SNMP transport protocol (see Figure 2.2). That decision was made mainly due to the scalability reasons. Namely, being centralised, SNMP would impose a huge demand on the manager platform system resources to be able to accommodate many open TCP connections to the managed devices. UDP also has small footprint on network resources compared to TCP, while being well suited for short request/ response type of operations, which is consistent with the connectionless nature of SNMP.

In general, the SNMP framework provides much poorer functionality and expressiveness than CMIP. However, because of the overall complexity and size of CMIP, many claim that this is a case of the cure’s being worse than the disease. These problems and most importantly the domination of Internet over OSI have prevented CMIP from reaching the dominant market position that was originally anticipated. On the other hand, the inherent simplicity of SNMP has been the driving force for its wide acceptance and popularity.

2.4.1. SNMP Protocol Data Units

SNMP uses relatively simple operations and a limited number of Packet Data Units (PDU) to perform its functions. Figure 2.3 shows the types of messages exchanged between the manager and the agent. Five PDUs have been defined in the first version of the standard (SNMPv1) [CAS90]:

- *Get Request*: it is used to access the agent and obtain managed objects values. It includes identifiers to distinguish it from multiple requests.
- *Get-Next Request*: it is similar to the Get Request and permits the retrieval of the next logical identifier in a MIB tree.
- *Set Request*: it is used to change the value of a MIB object.
- *Response*: it responds to the Get, Get-Next and Set Request PDUs. It contains an identifier that associates it with the PDU it responds to. It also contains identifiers to provide information about the status of the response (error codes, error status and a list of additional information).
- *Trap*: it allows SNMP agents to report events at their local NE or to change the status of the NE.



PDU Type	request-ID	0	0	varbind list

(a)

PDU Type	request-ID	error-status	error-index	varbind list

(b)

PDU Type	request-ID	non-repeaters	max-repetitions	varbind list

(c)

name$_1$	value$_1$	name$_1$	value$_1$	…….	name$_1$	value$_1$

(d)

Figure 2.3: SNMP PDU formats: (a) Get request, Get-Next request, Set request, Trap, Inform, (b) Response, (c) GetBulk request, (d) varbind list

Management data are returned in a list structured as a sequence of <object ID: value> pairs, termed the *varbind* list. As shown in Figure 2.3, SNMP request and response messages have for simplicity reasons the same packet format. Later protocol versions (SNMPv2, SNMPv3), define two additional operations [STA99]:

- *GetBulk Request*: it has been devised to minimise the number of protocol exchanges required to retrieve large volumes of management data, although there is a maximum PDU size limitation. It includes a field the specifies the number of variables in the varbind list for which a single lexicographic successor is to be returned (`non-repeaters`) and another field denoting the number of lexicographic successors to be returned for the remaining variables (`max-repetitions`).

- *Inform Request*: this PDU is used for manager-to-manager communication, i.e. it is sent by an entity acting in a manager role on behalf of an application, to another entity acting in a manager role, to provide information to an application using the latter entity.

2.4.2. SNMP versioning history

Two of the reasons for the initial success of SNMPv1 have been its lightweight design compared to OSI management and also the fact that it avoided the four-year standardisation cycles of the ITU-T [MAR00]. Yet, experience has shown that SNMP evolves at an even slower pace. The phenomenal success of the SNMPv1 architecture has also been the cause of its decline. It proved to be good for managing relatively small networks, but could not scale to large networks (e.g. geographically dispersed enterprises), and could not cope with large volumes of management data [MAR99b]. The telecommunications world had already shown how to solve this problem: by distributing the load across a hierarchy of managers. But strangely enough, management distribution was not a priority at the Internet Engineering Task Force (IETF) until the late 90s. Since SNMPv1 (standardised in 1990), four management architectures have been released: SNMPv2p, SNMPv2u, SNMPv2c, and SNMPv3 [STA99].

The first three only support centralised management. SNMPv2p has been rendered obsolete by the IETF in 1996 [PER97, STA99]. SNMPv2u had little success and “saw no significant commercial offerings” [PER97]; it is thus no longer used. SNMPv2c is often used to manage busy backbone routers, because it supports 64-bit counters and offers better error handling than SNMPv1; but it brings nothing new as far as distribution is concerned. As for SNMPv3, its main focus is on security [STA98], not scalability. In effect, it took eight years before the IETF delivered a substantial new release, SNMPv3, and another two years before major vendors began supporting it[2]. Its use is thus expected to remain marginal in production environments in the foreseeable future. Note that the MIBs adding support for one kind of delegation in SNMPv3 were issued only in 1999 [LEV99], so it will take even more time before they are implemented and deployed. In short, vendors of SNMP management platforms are currently forced to resort to proprietary extensions to support hierarchies of managers [MAR99a].

2.4.3. Strengths/Limitations of SNMP

According to the main characteristics of SNMP described in the previous section, the main strengths and contributions of SNMP-based management are the following:

- interoperability;
- simplicity;
- wide support by IP-equipment vendors;
- small footprint on agents;

On the other hand, SNMP also exhibits several weaknesses:

(I) Scalability

Scalability issues can be classified into the following categories:

Network overhead: In the context of NSM, network overhead is the proportion of a link capacity used to transfer management data, and thus unavailable for user data. The purpose of a network is to transfer user data, not management data, so an important goal of NSM is to keep network overhead low. SNMP is characterised by high network overhead, which is mainly due to the *polling*-based nature of monitoring and data collection process (pull model) [GOL91]. The manager repeatedly requests and retrieves specific MIB object values at each poll cycle from remote agents. In many monitoring applications, a considerable portion of network bandwidth is typically wasted to learn nothing other than that the network is operating within acceptable parametrical boundary conditions.

Latency: For polling, latency is the time elapsed between the moment the manager requests the value of a MIB variable and the time it receives it from the agent. It is important to keep latency reasonably low, so as to quickly detect and correct operational problems. End-to-end latency depends on networking conditions (the capacity and error rates of links, the speed of the IP routers traversed between the agent and the manager, etc), the amount of retrieved data and the number of protocol exchanges. When large sets of NEs need to be managed through SNMP, latency can very be high, especially when the ‘control loop’, i.e. the network distance, between the managing and the managed entities is large.

Manager’s processing capacity: The manager’s hardware resources (CPU, memory, etc.) dedicated to management applications cannot be continuously increased, due to cost and hardware constraints, setting a limit on manager processing capability [MAR98]. The centralised structure of SNMP architecture and the lack of data filtering capability characterising SNMP agents result in transferring vast amounts of NSM data, subsequently processed at the manager platform. This forces manager’s processing capacity to its limits and intensifies the need to relieve the manager from performing routine data processing tasks.

[2] For instance, Cisco supports it as part of IOS 12.1.5, officially released in December 2000 [Cisco00].

Capacity of the manager's local segment: The management data sent by all the agents converge toward a single point, the network segment where the manager is connected to, inevitably creating a bottleneck [BAL97].

Inefficient bulk management transfers: As the amount of data to transfer grows, it makes sense to reduce the overhead by sending the data in bulk, that is, to send unlimited number of MIB variables at a time, while keeping the number of network interactions low. However, SNMP has not been designed for transfers of bulk management data (typically stored in SNMP tables[3]). In SNMPv1, tables are retrieved through successive `get-next` operations. If the table includes many rows, the manager must perform at least one `get-next` per row [SPR99]. For tables with hundreds or thousands of rows, an equal number of requests/responses will be transferred through the network increasing the network overhead and latency. The situation improves with the `get-bulk` operator offered by SNMPv2c and SNMPv3 frameworks [STA99], which allows transferring more data per SNMP message. However, the manager should guess the length of the table to be retrieved and accordingly choose a value for the `max-repetitions` parameter. Using a low value will cause more PDU exchanges than necessary. Using a high value, however, can result in an *overshoot* effect [SPR99]: the agent can return data of no interest for the manager. The problems associated with bulk management data retrieval are elaborated in Chapter 7, which proposes MA-based applications to address all these problems.

SNMP message maximum size: The large number of protocol exchanges required to complete bulk management data transfers is mainly due to the maximum size limit of SNMP messages. All SNMP agents must accept SNMP messages that are up to 484 bytes in length, but may legally refuse longer messages [CAS96]. Yet, many open-source implementations of SNMP have used the maximum size limit of 1472 bytes (in LAN environments), proposed in [ROS96]. Clearly, when transferring data in the order of Mb, a large number of PDUs will be exchanged; the exact number will depend on the maximum size limit used in the particular SNMP implementation.

Poor efficiency of BER encoding: The SNMP protocol uses the Basic Encoding Rules (BER) [ITU94] to encode management data prior to sending it over the network. BER encoding can be implemented with very compact code (small footprint on agents) and causes a reasonable overhead on the agent and manager for encoding/decoding. Yet, the amount of administrative data (identifier and length) is large compared to the payload (content). This makes the network

[3] Simple, two-dimensional tables are the only form of structuring data in the SNMP architecture [STA99].

overhead unnecessarily large. It also increases the end-host latency, because more data take more time to transmit [MIT94].

OID naming scheme: This relates to the information model of SNMP-based management, i.e. the naming conventions for MIB variables. The Object Identifiers[4] (OID) transferred in SNMP messages exhibit a high degree of redundancy. For instance, all objects stored in MIB-II[5] are prefixed with 1.3.6.1.2.1. If this prefix could be omitted, a significant proportion of the space dedicated to the OID name would be saved. Furthermore, the prefixes of the table object OIDs are all identical up to the column number. In this case, more than 90% of the OID name is redundant [SPR99]. All these observations indicate a highly inefficient OID naming scheme.

No compression of management data: The first two versions of SNMP (v1 & v2c) did not allow the transparent compression of management data in transit. This unnecessarily increases network overhead and also network latency due to transferring larger volumes of data. As of SNMPv3, it is possible to compress management data by adding encryption envelopes to SNMP messages [SPR99]. Although this feature was initially intended for encrypting data, it also allows for data compression. When large chunks of data are compressed, the overall latency is also reduced, as the compression time is typically negligible with respect to the time saved to transmit the uncompressed data.

II) Unreliable transport protocol

Another problem of SNMP-based management is the transport protocol used for transferring SNMP messages, UDP. UDP operates in a connectionless fashion, which saves the three-way-handshake overhead of TCP [TAN96], and is ideal for exchanging short messages. However, due to the lack of acknowledgements, it is not suitable for communicating critical notifications to the manager. By using an unreliable transport protocol, the management system runs the risk of losing important notifications for trivial reasons such as buffer overflows in IP routers [MAR99b].

III) Security

SNMPv1 and v2c adopt a *weak security scheme*. Passwords for configuring routers, hubs and servers are passed across the network as clear text, in unprotected packets. Identification, which is based on *community strings*, is so simplistic that cannot be considered as secure. The main advancement brought by SNMPv3 is security. SNMPv3 supports identification,

[4] An OID (Object IDentifier) uniquely identifies a MIB object.

[5] MIB-II [McC91] is the standard MIB in the IP world, supported by virtually all SNMP-compliant network devices and systems.

authentication, encryption, integrity, access control, etc [STA98]. However, the support of SNMPv3 by major vendors takes place at a slow pace.

IV) Information model: low level of semantics

As far as semantic richness is concerned, the main shortcomings in SNMP are the absence of high-level MIBs, the limited set of SNMP protocol primitives, and the data-oriented nature of the SNMP information model. Due to these limitations, developing high-level management applications is a difficult task, which partly explains why NSM applications are often limited to little more than monitoring in the IP world [MAR00]. In particular, due to the way the SNMP market evolved over time, SNMP MIBs offer only low-level Application Programming Interfaces (API), often called *instrumentation MIB*s. SNMP frameworks provide no support for management applications to dynamically define external data models as part of the MIBs. Although it is possible for applications to retrieve raw MIB data and compute the appropriate data model at the platform host, this is highly inefficient [GOL95]. In addition, SNMP protocols support limited protocol primitives vocabulary, which allows getting/setting atomic variable values and notifying about important events. This is extremely restrictive and is typical of a data-oriented information model, unlike object-oriented models, which are widely used in industry today. The absence of an object-oriented information model in SNMP is generally regarded as one of the main limitations of SNMP [GOL96]. When the Distributed Management Task Force (DMTF) endeavoured to define a new management architecture in the late 90s, it came as no surprise that its first delivery was a new object-oriented information model: the Common Information Model (CIM)[6] [BUM00].

The *scalability* problems of SNMP are mainly attributed to its centralised model and can be efficiently addressed through MA-based approaches. The latter enable the dynamic delegation of NSM functionality to managed elements, where MA objects may filter/correlate management data, adopt an event-driven (instead of polling-based) approach to notify managers about important events and apply data compression, thereby reducing network overhead. Chapter 7 describes MA-based monitoring applications that address these issues. In addition, as described in Section 4.4.1.2, the administrator is given the option to use *either TCP or UDP* for MA migrations, depending on the application reliability requirements. Our model also addresses the authentication, authorisation and encryption *security* aspects (see Section 4.4.1.3.2). Finally, *semantically rich* management operations can be built using low-level

[6] CIM is defined as a conceptual information model for describing management that is not bound to a particular implementation. This allows for the interchange of management information between management systems and applications. This can be either ‘agent to manager’ and ‘manager to manager’ communications which provides for Distributed System Management.

protocol primitives as a basis and dynamically deployed to managed systems. This issue is discussed in Chapter 7.

2.5. Distributed Objects-Based Management

A new programming paradigm has emerged in the ’90s, Distributed Objects Technologies (DOT). DOTs define an object-oriented paradigm, where objects can interact even if they do not reside on the same system. Since OSI-SM is object-oriented and SNMP managed objects can be mapped onto objects, it took little time for NSM researchers to start working on the integration of DOTs with existing management architectures. Thus, DOTs enabled a paradigm shift from the protocol-based approaches of the early 90’s, exemplified by the SNMP and OSI-SM, to distributed object-based approaches in the mid to late 90’s. In this section, we briefly describe two representative DOTs, the Common Object Request Broker Architecture (CORBA) and Java Remote Method Invocation (RMI), discussing their relevance to NSM.

2.5.1. CORBA

CORBA [CORBA] is the product of a consortium of over eight hundred companies, known as the Object Management Group (OMG). The OMG has approached the problem of handling the interaction of distributed components by creating interface specifications, and *not* code. Distributed components of the system are able to describe their interfaces using the Interface Definition Language (IDL) and subsequently inter-operate through the underlying Object Request Broker (ORB). Namely, the ORB provides the communication backbone through which distributed components are able to interact. To perform requests or return replies, objects use a generic RPC-like request/response protocol, the General Inter-ORB Protocol (GIOP) or its TCP/IP mapping, the Internet Inter-ORB Protocol (IIOP). Distributed components communicating via an ORB do not need to be aware of the mechanisms used in that communication and are able to discover each other at run time. A number of CORBA *services* provide basic functions, e.g. the *Naming* service that allows clients to locate objects based on their names, the *Trading* service that enables objects location based on their properties and the *Event* service which allows asynchronous messaging between objects.

The applicability of distributed objects on NSM has been a subject of intense research in the past few years [MAZ96]. Along this line, Pavlou pointed out the suitability of DOTs in large scale distributed environments and proposed the use of CORBA in TMN open interoperable interfaces, replacing OSI-SM [PAV00]. However, a main direction of the research efforts has been on the seamless integration of legacy systems into emerging distributed object environments. In that context, Mazumdar proposed the use of a gateway, which achieves the inter-operation of management applications in CORBA domain and agents in SNMP domain [MAZ96]. The main function of the CORBA/SNMP gateway is to dynamically convert method invocations on object references in CORBA domain to SNMP messages for MIB entries at remote agents. Likewise, a CORBA/CMIP gateway has been proposed in [CHA97], which hides the complexity of OSI management from developers.

2.5.2. Java RMI

Java RMI [RMI] of Sun Microsystems is an API defined for remote communication by the Java development team. It allows method invocations between objects residing in different address spaces in a seamless and location-transparent way. This facility requires that the remote objects implement a specified interface and that their hosting devices run a dedicated service (*rmiregistry*).

However, Java RMI technology provides a low-level communication infrastructure. The syntax and the content of the messages are not defined (or remain in a very poor form). Another major drawback of RMI is that unlike CORBA, it is *not* language-independent, but exclusively intended for distributed Java-to-Java applications communication [MOR97]. RMI uses the object *serialisation* (described in Section 3.5.2) facility of Java to *marshal* and *unmarshal* parameters. Methods invoked at remote objects are reachable through the RMI's transport protocol, the Java Remote Method Protocol (JRMP), which has its own lookup service. In the context of NSM, Java RMI has been used in the JMAPI and JMX initiatives of Sun Microsystems (see Section 2.7.2).

A third emerging DOT is Microsoft's proprietary solution, the Distributed Component Object Model (DCOM) [DCOM], which has not yet found wide acceptance in NSM applications development. This is due to its limited platform support (Microsoft platforms) and the fact that DCOM is still an immature technology, in comparison with CORBA.

2.5.3. Limitations

Despite the undeniable suitability of DOTs for building distributed management applications, a number of disadvantages have been identified in the literature. The first three are specific to CORBA and RMI, while the last generally applies to all existing DOTs:

- Even though a main aim of distribution is scalability, CORBA retains a reliance on centralised information stores for such things as the name service, the trading service and implementation repositories. Although that has the benefit of simplicity, the centralisation of data represents a potential performance bottleneck and a single point of failure [McK00].

- The communication protocol of CORBA has been criticised for its efficiency, as it gives rise to a high message overhead on certain operations [GOK98] and is slower than traditional Remote Procedure Calls (RPC) [LIP00].
- Distributed applications relying entirely on Java RMI have been criticised for low scalability due to the limitations of JRMP (low protocol speed, and out-of-control socket creation). However, the situation improves in the latest releases of RMI that support the IIOP transport protocol, which offers improved scalability, while allowing the inter-operation of RMI and CORBA-enabled applications [MOR97].
- In DOT-based approaches, the functionality of the distributed objects is static and cannot be altered, in a similar way to OSI-SM and SNMP support object facilities [BOH00b]. The required degree of flexibility and reconfigurability can be achieved through the mobile code paradigms described in Chapter 3, which allow the dynamic deployment of management services at runtime.

2.6. DIRECTORY ENABLED NETWORKS

The emerging Directory Enabled Networks (DEN) [STR99] framework signifies a shifting from DOT-based approaches back to protocol-based approaches (exemplified by the SNMP and OSI-SM architectures). Like OSI-SM, DEN uses OSI directory technology to store information about networks. Changes in network configuration are handled within directories, which store information hierarchically, with each entry (object) containing a set of attributes. DEN directories distribute management information across multiple machines through a process known as *directory replicatio*n. DEN directories are typically accessed via the Lightweight Directory Access Protocol (LDAP), which supports three types of protocol operations: query, update, and authentication. The purpose of DEN framework is to facilitate the management of large and complex configurations of heterogeneous systems and deal with dynamic networks with frequent changes in their configuration. Therefore, DEN tackles the dynamism and scalability challenges in network management.

2.7. JAVA-BASED MANAGEMENT

Another approach that emerged in the 90's has been based on Java [JAVA], which in the NSM world is considered as a *technology* rather than as a programming language. This section starts with an overview of Java, highlighting its generic advantages and deficiencies (especially those with relevance to NSM applications). This overview is necessary, as it justifies the selection of Java as the implementation platform for our MA-based management framework. A state-of-the-art report on Java-enabled management technologies follows, along with an overview of research initiatives undertaken in Java-based NSM. For further information on research activities and commercial products related to Java-based management, the interested reader is referred to the links given on [JBM].

2.7.1. Overview of Java

Since its first release in 1995 by Sun Microsystems, Java has become one of the most popular development platforms within the Internet community, mainly due to its strong connection with Web-based applications, inherent portability, simplicity and support for Internet-oriented programming. Java is described as "*a simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language*" [JAVA].

2.7.1.1. Generic Benefits & Deficiencies of Java

Architecture-neutral and portable: Java offers the advantages of architecture neutrality and portability to networking application developers, which led to the Sun's marketing phrase "write once, run anywhere". That allows developing applications that address the problem of increased heterogeneity characterising modern networks, in terms of operating systems and hardware architectures. To achieve platform-independence, the Java compiler does not generate "machine code", i.e. native hardware instructions, but *bytecode*: high-level, interpreted, portable code. The format of this binary code is architecture-neutral. Given that the run-time platform, i.e. the Java Virtual Machine (JVM), is available for a platform, any Java application can execute there.

Object-orientation: Object-oriented design is a very powerful programming paradigm as it facilitates the clean definition of interfaces and maximises software re-usability. Although there are several object-oriented languages, Java is the only one that *enforces* it, i.e. the programmer has no choice but to encapsulate all data in objects.

Multithreading: In most languages, writing programs that deal with many processes (*threads*) simultaneously can be a very difficult task. Built-in multithreading support is one of the most powerful features of Java, as it allows lightweight process concurrency. The Java Development Kit (JDK) library provides a rich collection of methods to start, stop, suspend, resume or check the status of a thread.

Strong networking support: Java is particularly suitable for network programming. It offers an extensive library of classes and routines, which include TCP and UDP socket communication, support for broadcast and unicast messaging, invocation of remote methods through RMI, etc.

Native methods: When developing an application, situations might arise that necessitate the integration of code written in other languages (*native code*) within the Java application. The use of native code might be required for one of the following reasons:

(i) Code re-usability: Most of existing software is not written in Java. In some occasions, it is time consuming to translate it to Java, so the direct integration of non-Java with Java front-end programs might be preferable.

(ii) Performance: Being an interpreted language, Java exhibits worse performance compared to other languages that involve compiled code. Hence, native code could be more suitable for time-sensitive applications, in order to maximise the execution speed.

(iii) Tasks that cannot be implemented in Java: In certain cases, the implementation of certain tasks in Java might not be feasible (e.g. when low-level, platform-dependent functions have to be used), making the need for co-operation with native code imperative.

The integration of Java programs with native code is achieved through the Java Native Interface (JNI) [JNI]. An important feature of JNI is that it allows ‘bi-directional’ communication between Java applications and native code, i.e. Java methods may invoke native functions and vice-versa.

Security: Java is intended for use in networked/distributed environments. Along this line, particular emphasis has been placed on security. A built-in security mechanism is provided with the core JDK API: the `java.lang.SecurityManager` class that developers may extend to customise the access rights of individual applications on system resources [AUS00]. In addition, the Java Security API [JSAPI] is a framework for developers to easily include security features in their applets[7] and applications. The Security API includes support for digital signatures, encryption and authentication.

Despite all its advantages, Java has been often criticised for performance-related issues. The fact that Java is an interpreted language makes its performance substantially poorer than compiled languages, such as C++. However, with the advent of Just-In-Time (JIT) compilers [JIT], Java programs may be executed much faster. JITs basically compile the bytecode into platform-dependent native code, which is subsequently executed (it is faster to read the bytecode, compile it and run the resulting executable, than it is to interpret it).

[7] Applets are Java programs that can be included in an HTML page and subsequently uploaded and executed by Java-enabled Web browsers JVMs.

2.7.2. Java Technologies for Network & Systems Management

The self-evident suitability of Java for developing management applications has been recognised since its early days, with the first commercial implementations of the SNMP stack in Java released in 1996 [AdventNet]. Since then, AdventNet, a company that specialises in Java-based management products, has gone much further providing a complete suite of Internet management tools. These include a visual builder tool used to build SNMP management Java applets and applications, an Agent Toolkit that automates the process of creating SNMP MIBs and instrumenting agents for these MIBs, etc.

The following section summarises the technologies recently emerged in the field of Java-based management. All these technologies represent initiatives undertaken by Sun Microsystems and comprise toolkits (JMAPI and JMDK) or standards (JMX) expressly oriented to building management frameworks.

JMAPI - JDMK - JMX

In 1996, shortly after the release of the JDK 1.1 that added support for RMI, Sun Microsystems made the Java Management API (JMAPI) [JMAPI] publicly available. This API is a set of tools and guidelines to build management applets supporting RMI. It supports the most common SNMP MIB, MIB-II [McC91], by mapping its managed objects onto Java objects. It also provides a rich graphics library to ease the development of sophisticated GUIs, allowing the visual representation of management information.

Soon after, Sun released the Java Dynamic Management Kit (JDMK) [JDMK], a component-oriented management toolkit written in Java. Like JMAPI, JDMK is publicly available. It is based on Java Beans (JB)[8] and comes with a library of core management services. It also contains adapters to enable communication via RMI, HTTP and SNMP. Unlike JMAPI that deals only with MIB-II, JDMK includes an SNMP-to-Java MIB compiler, which translates the managed objects defined in any SNMP MIB into JB components (called management beans, or MBeans). The toolkit supports push and pull from agents and offers a powerful framework for developing management applications.

In 1999, JMAPI was superseded by the Java Management eXtensions (JMX) [JMX]. JMX is a management framework intended for object-oriented Web-based management. It is far more comprehensive than JMAPI and builds on the experience acquired by Sun with the JDMK. The specification does not only focus on the agent part of the management system (as it is the case with JDMK) but also specifies the manager part. In other words, JDMK can be considered as an integral part of JMX. It should be noted though that being relatively new

[8] A JavaBean [JB] is a reusable component that can be visually manipulated by means of a builder tool.

technologies, JDMK/JMX need extensive evaluations and performance studies to test their performance and scalability.

2.7.3. Research approaches to Java-based management

In addition to commercial activities, the power of Java in building management applications has been also signified by the profound interest of the research community. Early approaches focused on developing centralised frameworks entirely built in Java [LUD97, PAR98]. Both these works introduced a multi-threaded Java Agent engine, and proposed the replacement of BER encoding by a heavyweight bytecode-based mechanism. In particular, a simple `get` request involves the creation, compilation and transfer of a Java class that encloses the requested OID string, with the receiving Agent loading the class and retrieving the OID string before returning the requested value using the same mechanism. This method also imposes heavy network traffic (unnecessary transfers of classes) and exhibits the drawback of delay imposed by the time-intensive processes of compilation and class loading. These problems were addressed by a paper describing preliminary work of the author, which introduced a lightweight socket-based communication mechanism [GAV99].

More recent approaches to Java-based management concentrate on encompassing the Sun's standards, described in the preceding sections, in distributed NSM frameworks. For instance, [KEL99] described a case study for dynamic management of Internet telephony servers based on JBs and JDMK. Lee described the design and implementation of a management platform based on the TMN and discussed how Java technologies can support a variety of management interfaces as service components in a distributed computing environment [LEE00]. Anerousis introduced *Marvel*, a sophisticated Java-based management framework that enables the development of scalable NSM services [ANE99]. Scalability in Marvel is achieved by supporting computed views of low-level management information that convey high-level network operational status statistics and distribute the view computation task to a hierarchy of processors (servers). Information stored in Marvel objects is accessible through a Web interface.

2.8. WEB-BASED MANAGEMENT

Java owns its popularity mainly to its strong connection with the Web through applets. Since the Web is now ubiquitous, several proposals (see references in [Web]) have been made to use the Web technology in NSM. Wellens and Auerbach introduced the concept of *embedded management applicatio*n, where an applet is stored in the managed device and loaded by the administrator into a Web browser; communication between the applet and its origin agent later relies on HTTP instead of SNMP [WEL96]. Since the time of this proposal, new technologies, such as Java *servlets*[9] and RMI have appeared and been used in Web-based management, for instance to open persistent sockets between applets and servlets, etc. Recent approaches realised a step towards decentralisation through the transition from *pull* to *push* management [MAR99b, ADA00]. In the push model, management data transfers are always initiated by the agent, similarly to SNMP notifications, thereby reducing the network overhead, and moving part of the CPU burden from managers to agents [MAR99b].

In the standardisation arena, an industrial consortium led by Microsoft launched a new initiative in 1996: the Web-Based Enterprise Management (WBEM) [WBEM]. WBEM took a revolutionary approach by replacing all existing protocols and object models with new ones. The main motivation was the integration of the Desktop Management Interface (DMI), used to manage cheap desktops, with SNMP, used to manage network equipment and expensive workstations. In this context, a new object model, CIM, has been devised. Today, SNMP/CIM, DMI/CMI and CMIP/CIM gateways are under development. WBEM is backed by most vendors in the NSM industry and is likely to emerge as one of the main management architectures of the decade [MAR00].

2.9. THE NEED FOR DISTRIBUTED MANAGEMENT

Management world today is dominated by protocol-based approaches, exemplified by the SNMP and OSI-SM. Both the IETF and the OSI approaches are characterised by the inflexible manager-agent paradigm. Centralisation has a serious impact on management scalability since it imposes almost all the computational burden on the manager platform [BAL97, CHE98]. The operations available for accessing MIBs are very low-level. In SNMP, for instance, the manager can only get and set atomic values in a MIB. This fine grained CS interaction is often called *micro-management* [GOL91], and leads to the generation of intense traffic and processing bottlenecks.

The OSI-SM supports the delegation of monitoring activities to the NEs, reporting only QoS alarms or summarised reports to higher-level managers [OSI92, OSI93]. Nevertheless, such generic functionality needs to be first researched, standardised, implemented and eventually deployed to NEs; this process typically takes a long time. Furthermore, the same research-standardisation-implementation-deployment cycle needs to be repeated whenever any

[9] Servlets are server-side Java components; while applets provide a way of dynamically extending the functionality of client-side browsers, servlets allow the application developer to dynamically extend the functionality of network servers.

modification, e.g. for providing more sophisticated features that were not thought out in advance, is to be introduced [MOU98b, BOH00b].

The scalability problem of centralised architectures becomes more profound as the dimension of the managed network grows. The managers need to communicate with a larger number of devices, as well as store and process an ever-increasing amount of data. This leads to the need for high cost hardware dedicated to the manager platforms [MAR98] and poor performance. In addition, the network area around the manager stations is saturated due to the combination of messages sent by the management platforms with those sent by the devices. The worst shortcomings of the centralised approach show up during periods of heavy congestion, when management intervention is particularly important [PIC98, KIM98]. During these periods: (i) the manager increases its interactions with the devices and possibly downloads configuration changes, thereby increasing congestion, (ii) access to devices in the congested area becomes difficult and slow (sometimes even impossible), and (iii) congestion, as an abnormal status, is likely to trigger notifications to the manager, generating even more traffic. In order to answer the problems related to centralisation, NSM functionality must be *distributed*, i.e. complex diagnosing and information gathering activities must be moved from the managers to the managed devices.

The advantages of distributed management can be found in many research papers (e.g. [GOL91, BAL97, KAH97, CHE98, MAR99a]). In particular, management distribution:

- allows applications to efficiently exploit the increased availability of hardware resources of modern managed systems;
- improves the autonomy and survivability of NMSs. That is, when the communication with the managing process is lost, distributed management entities can continue to execute;
- reduces the need for intensive polling;
- reduces the computational load on manager platforms through pre-processing management data, filtering unimportant alarms and delivering only high-level information;
- leads to significant reduction of management traffic, as most management interactions are locally executed.

It should be emphasised though that *not all management applications should be necessarily decentralised*. In fact, centralisation is the appropriate model for applications that have little inherent need for distributed control. Such applications (a) do not require frequent polling or high frequency computation of MIB deltas, i.e. aggregation functions, (b) have high-bandwidth network connections between the manager and the managed devices, (c) exchange relatively small amounts of data, and (d) do not need frequent, semantically rich conversations between manager stations and managed nodes [MEY95] (see Figure 2.4).

CENTRALISED PARADIGM	**DISTRIBUTED PARADIGM**
Low need for distributed processing, e.g. a small network that can be managed by a centralised control system.	High need for distributed processing, e.g. localised information processing required either due to data size or for robustness.
Low frequency for required polling, e.g. the NMS does not require constant polling or the network has high bandwidth availability.	High frequency for required polling, e.g. need for constant monitoring of a large number of managed objects.
High ratio of throughput to the amount of management information.	Low ratio of throughput to the amount of management information.
Low need for semantically rich/frequent communication, e.g. networks that support simple services.	High need for semantically rich/frequent communication, e.g. large networks that support complex services.

Figure 2.4. Management centralisation or distribution metrics (adapted from [MEY95])

2.10. Decentralisation Initiatives within the Internet Community

The disadvantages of centralised management, as outlined in the preceding section, have first been admitted by the same organisations that introduced it. The following sections review the decentralisation initiatives undertaken by the IETF, with focus on IP networks management. The reader interested in extended surveys on the various approaches on management distribution may refer to [KAH97, MAR99a].

2.10.1. Management Distribution within the SNMP Frameworks

A primitive form of decentralisation (provided in SNMPv1) is the asynchronous notification mechanism. Namely, SNMP agents can send *traps* [CAS90] to the manager platform not as a result of a request, but when an important event occurs. Still, no management action can be performed locally as decisions are made centrally by the manager application.

The SNMPv2 introduces the concept of *proxy agent* [McC96], which realises a transition from centralised to hierarchical management models (see Figure 2.5). A proxy agent can be responsible for a set of devices; the manager sends requests to the proxy instead of interacting directly with these devices. Traditionally, SNMP has used proxy agents in a pass-through role, wherein a proxy passes the manager requests and agent responses through, in an essentially transparent mode. The fact that a proxy can be used as an intermediate manager for hierarchical management is recognised by SNMPv2, however, the framework provides no means for managers to delegate tasks to intermediate managers or to communicate with them during the execution of these tasks [KAL97]. SNMPv2 has also attempted to promote management distribution through introducing the concepts of the *inform* PDU primitive (see Section 2.4.1) and the Manager-to-Manager (M2M) MIB [CAS93], which however proved unworkable in practice and have now become obsolete [MAR00].

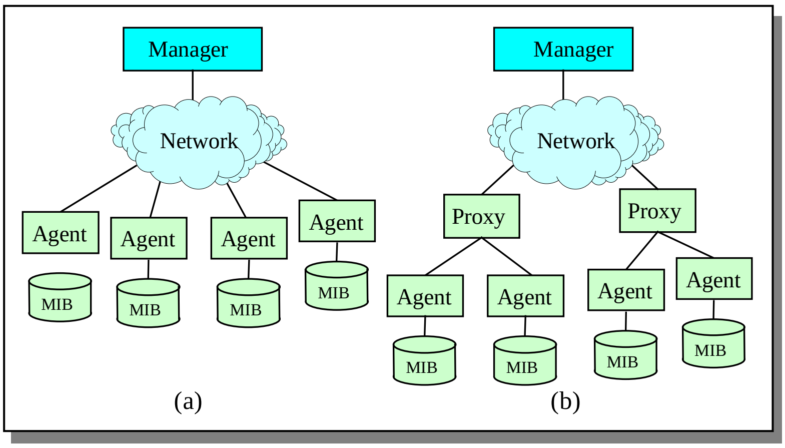


Figure 2.5. (a) Centralised management, (b) Hierarchical management

2.10.2. Remote Monitoring

The IETF has proposed another approach, known as Remote MONitoring (RMON) [WAD95], that introduces a higher degree of decentralisation. RMON assumes the existence of network monitoring systems called *monitors* or *probes*, which can either be standalone devices dedicated to link monitoring or embedded into network devices (Figure 2.6). By monitoring packet traffic and analysing the headers, probes provide information about links, connections among stations, traffic patterns, and status of network nodes. SNMP is used for communication between the manager and the agents running on probes. RMON allows the delegation of monitoring functions from the managers to the probes through the definition of suitable *filters*. A probe can detect failures, misbehaviours, and identify complex relevant events even when not in contact with the management station. In addition, the agent on the probe can perform semantic compression of data by pre-processing the information collected, before sending it to the management station.

However, RMON also exhibits several deficiencies:

- Typically, a stand-alone RMON compliant device (probe) is required to monitor the traffic activity of a single network segment, leading to considerable increase of cost when the management of multiple segments is required.

- The control operations of a RMON probe may be set/modified only at *configuration* time, i.e. *runtime* modifications are not supported [MOU98a].
- RMON is adequate for providing only *traffic-oriented* statistics since the status of the network is determined by direct inspection of the packets flowing in it, rather than inspection of the devices status, like in the mainstream (centralised) approaches that offer *device-oriented* statistics. Hence, RMON is not adequate when management operations should be applied in both system and network level.

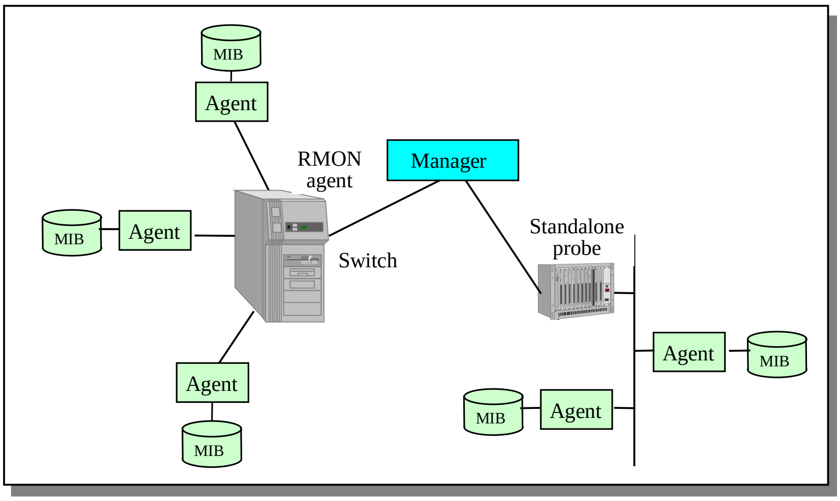


Figure 2.6. The RMON approach

2.10.3. Script MIB

Management distribution support to SNMPv3 framework has been added in 1999. In particular, the DISMAN (DIStributed MANagement) Working Group of the IETF was chartered to define an architecture where a main manager can delegate control to several distributed management stations. Among others, the DISMAN framework provides mechanisms for distributing scripts, which perform arbitrary management tasks to remote devices. This is achieved through the Script MIB [LEV99], which defines a standard MIB for the delegation and invocation of management functions (scripts), based on the Internet management framework (see Figure 2.7). According to the specification, the term script is very broad, referring to some type of executable code which can be executed by any device that implements the MIB.

In particular, Script MIB provides the following capabilities [SCH00]:

- Transfer of management scripts to a distributed manager;
- Initiating, suspending, resuming and terminating management scripts;
- Transfer of arguments for management scripts;

- Monitoring and control on executing management scripts;
- Transfer of the results produced by running management scripts.

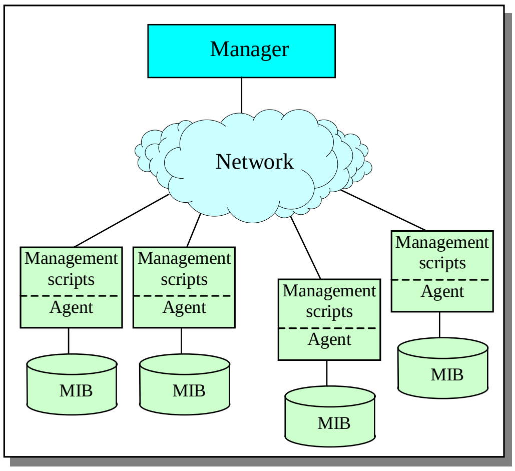


Figure 2.7. The Script MIB approach

The Script MIB is limited to the operations performed on the six tables that constitute the MIB. Before the administrator decides to delegate a script, he/she should first check the languages supported by the Script MIB implementation and select an appropriate script from a repository. Scripts are uploaded to the devices through client pull or server push models. The Script MIB enables scripts to be initiated, controlled and terminated through the SNMP management framework. For instance, the execution of a script starts with a single SNMP `set` request. The manager can also obtain intermediate or final results generated and maintained at the agent.

Although providing a powerful management distribution mechanism, the Script MIB also exhibits a number of limitations:

- The current specification makes it difficult to update existing scripts [McM99];
- Currently, the number of Script MIB implementations (such as the *Jasmin* project implementation [Jasmin]) is still limited, namely this technology has not been widely tested/evaluated yet;
- The Script MIB approach is specific to the Internet management frameworks, similarly to all standardisation approaches undertaken by the IETF or in the context of OSI-SM [BOH00c]. In contrast, MAs can provide a more generic and framework-independent mechanism for the delegation of management functionality.

It should be noted that parallel work to the script MIB has been also reported in the in the context of OSI-SM [VAS97].

2.11. RESEARCH APPROACHES ON MANAGEMENT DISTRIBUTION

The advantages of distributed/hierarchical management over centralised approaches have motivated several research works on that field, described in the following sections.

2.11.1. Management by Delegation

The full potential of large-scale distribution over managed devices was first demonstrated by Goldszmidt et al. through the *Management by Delegation* (MbD) framework [GOL91], which set a milestone in NSM research. MbD realises a distributed management architecture that enables the execution of management tasks at the end nodes by dynamically delegating management functions to stationary agents ("*elastic*" processes). The elastic processes allow new control functionality to be uploaded in the form of scripts, using a proprietary delegation protocol, the Remote Delegation Protocol (RDP). Scripts execution is usually initiated by other processes that require the corresponding functional services. This is quite similar in concept with the Script MIB approach (see Section 2.10.3); in fact, the latter follows the MbD paradigm, while the same applies to most recent standardisation and research NSM distribution initiatives.

Epitomising, the MbD paradigm represents a powerful management distribution approach, which has promoted for the first time network devices from 'dumb' data collectors to the rank of full-fledged managing entities. MbD can be considered a precursor of the ideas introduced in this thesis. The fundamental difference lies on the fact that in MA-based NSM, delegation of management functionality is achieved through the MA objects and not downloadable scripts; the execution of MAs is not necessarily restricted on a single device, as they can autonomously migrate from host to host.

2.11.2. Flexible Agents

The work presented in [MOU98a] builds upon the groundwork of the MbD paradigm, introducing an intelligent agent module, termed the "*flexible agent*". Unlike MbD agents, flexible agents do not operate at the managed device level; instead, they are responsible for a number of standard SNMP agents located within their management domain and exploit their ability to communicate and co-operate with their peers to correlate/filter collected data. The applicability of the flexible agents approach has been demonstrated in fault management scenarios [MOU99].

2.11.3. The Spreadsheet approach

A *spreadsheet* scripting environment for SNMP is proposed in [KAL97]. The spreadsheet scripting language allows a manager to prescribe computations that can be carried out by the agent and supports arithmetic, logical and relational operators. Each cell in the spreadsheet defines an expression that computes a value from other given data such as values in the MIB. Expressions can be inserted, updated or deleted according to the manager needs, through SNMP operations. Apart from computed attributes, the spreadsheet is also capable of generating event reports by evaluating predicates containing relational expressions.

2.11.4. Hierarchical Management

A hierarchical NSM system that uses the concept of the *SubManager* has been presented in [SIE96][10]. A SubManager is responsible for few agents; it collects primitive data from them, performs some calculations and produces more meaningful values that can be used by a superior manager. This method significantly reduces the volume of NSM traffic since only high-level information is sent to the manager. A similar approach is adopted in [KOO95] that proposes the use of *Area Agents*, each performing local management to a specific network section. Area Agents can be configured to collect and pre-process all data that was formerly dealt with by the manager station.

The *Mid-Level Manager* (MLM) [MLM] of SNMP research has been another step towards management distribution. An MLM is a dual-role entity, i.e. it acts both as an agent and a manager. When managers request information from the MLM, it plays the role of the agent, while acting as a manager for the agents located within its domain. When the MLM is placed across a WAN link, remotely from the enterprise manager, it obviates to a certain degree the necessity of performing “normal” SNMP polling. This reduces the polling traffic over the WAN link, thereby achieving significant cost savings. Furthermore, should the WAN link “go down”, the MLM continues to perform management tasks at the remote site despite its separation from the central manager. Finally, MLMs can provide a prompt first-line response to problems while leaving the manager in the position of final authority. After the first-line response, the manager can evaluate the situation more thoroughly and decide whether further action is required or not.

[10] It should be noted that the concept of hierarchical management has been first proposed in the TMN recommendation [CCITT92].

2.12. Synthesis & Evaluation of Existing Approaches on Management Distribution

The approaches investigated in Sections 2.10 and 2.11 certainly offer useful mechanisms for realising management distribution. In general, they can be classified in:

(a) *Static* approaches (proxy agents, RMON, flexible agents and the hierarchical management models described in Section 2.11.4), which offer distribution but very limited flexibility;

(b) *Dynamic* approaches (MbD, Script MIB, spreadsheet approach), which offer distribution and increased degree of flexibility).

With the exception of RMON, static approaches achieve management distribution through building management hierarchies, where a mid-level delegation entity is responsible for managing a group of managed devices. That model is consistent with the hierarchical structure of modern large-scale enterprise networks, obviates the need for remote communication between the manager and managed elements and results in localisation of management traffic within the individual management domains.

There are two parts on building management hierarchies: (i) assigning roles ("proxy agents", "flexible agents", "area agents" "SubManager", "Mid-Level Manager") to the members of the hierarchy, and (ii) assigning members to specific physical locations where they will function under the supervision of higher-level members. This is feasible if the network is moderately small and/or not very dynamic, and if a single manager is the only user and specifier of the management policies. However, it is not in step with the dynamically evolving topological and traffic characteristics of large-scale enterprise networks [LIO01]. In addition, the distribution of aggregation and filtering computations in these approaches is manual and static. The manager of such a system is required to know the managed network well enough to build a hierarchy of distributed servers that will accommodate, to the best extend possible, all desirable computations that could be performed on NSM data. In general, the concept of management domain is not clearly defined in static approaches, nor are the criteria according to which the boundaries of such domains are determined. Also, the MLMs, SubManagers, etc, may suffer from overloads while executing their tasks [LOP00], either due to limited capacity or insufficient computing power of the hosting processors.

The problem of architecture inflexibility also applies to RMON where the filters configured by the manager are statically predefined and cannot be easily changed at runtime [KAL97, MOU98a].

The inflexible definition of static approaches can be addressed by MAs that dynamically migrate to remote management domains when certain conditions are satisfied and acting as mid-level management entities. In Chapter 6, we will introduce a hierarchical MA-based management framework that addresses the issue of dynamic deployment and placement of agents, operating at an intermediary level between the manager and the managed devices.

Dynamic approaches offer increased flexibility in comparison with their static counterparts, as they allow on-the-fly customisation of delegated functionality. However, they exhibit a number of limitations. First, they rely on *scripts* as a means for distributing management intelligence to end nodes. Such scripts are typically coded using script languages, such as Perl, Javascript and Tcl/Tk. Script languages allow rapid coding and are ideal for implementing programs with limited capability. On the other hand, MAs are typically programmed in high-level programming languages (e.g. Java), which are suitable for developing more complex and demanding decentralised management tasks [PUL00a].

In addition, the Script MIB and the spreadsheet approach both rely on SNMP for script transfers. As a result, they suffer from disadvantages related to SNMP, that is:

a) not guaranteed delivery: SNMP relies on UDP for message transfers, which makes use of an unreliable packet transport mechanism;

b) SNMP packet length limitation: scripts are typically downloaded line-by-line [McM99] to minimise the risk of exceeding the packet length limit, resulting in additional traffic and increased unreliability and latency, especially when downloading large scripts.

On the other hand, the MbD approach uses a proprietary protocol for uploading scripts (RDP) and has been criticised because its delegation primitives have not been integrated with the SNMP framework [KAL97]. We believe that the wide spread of SNMP should certainly be taken into account, when designing distributed management architectures. The desired compliance with SNMP framework does not, however, compel the use of SNMP as a delegation protocol, neither necessarily entails suffering from the generic limitations of SNMP frameworks highlighted in Section 2.4.3. Our MA-based management infrastructure, presented in Chapter 4, is fully integrated with SNMP frameworks, however it relies on more efficient and reliable protocols than SNMP for MA migrations.

A common idea shared among all dynamic approaches is to upload code down to the managed device level, thereby performing semantic compression of management data and reducing network overhead. This idea is also behind MA-based management approaches. However, existing dynamic approaches focus is on the entities that enable delegation, i.e. the delegation agents. This represents a conceptual difference with the MA paradigm, which focuses on the mobile entities themselves and not on the entities that create, dispatch and receive the MAs [CHE00b]. That is, the MAs can be viewed as mobile *managing entities* that migrate to specific managed devices on demand, to perform arbitrarily complex management tasks, with an increased degree of *autonomy*.

Through exploiting their *mobility* feature, individual MAs can also perform, if required, decentralised NSM operations over a group of managed elements. In that sense, MAs can be regarded as a ‘superset’ of delegation agents, as they can provide all the functionality offered by the latter, having the additional benefit of mobility. Mobility is used only when necessary or whenever it is more efficient in terms of management cost; should a static delegation agent is sufficient for performing a management function, an MA can be sent to managed device and remain there until no longer needed. In Chapters 5 and 7, we identify application scenarios where mobility is necessary and suggest ways to minimise the impact of MA transfers on network resources.

Another limitation of existing dynamic approaches is that they imply a *device-level* view of managed resources, since delegated code executes on a single-device [LOP00]. This problem is addressed by static hierarchical models, where mid-level entities can correlate data collected from their management area, providing a *domain-level* views, which can be useful to identify problems related to large groups of devices or aid in capacity planning. We believe that the “best of the two worlds” can be achieved through MAs that visit a set of devices and correlate the data collected from them to provide domain-level views of managed elements. This approach simplifies the data correlation process, as it obviates the need for a complex co-operation mechanism between delegation agents, as proposed in [MOU98a]. In Chapter 7, we describe an application that addresses this issue.

Furthermore, dynamic approaches only consider a one-to-one relationship between two entities, where one (the *delegator*) delegates some task to the other (the “*delegee*”); this task is executed under the control of the delegator. No aspects of *cascaded* delegation are supported, limiting the degree of distribution granularity[11] as the delegator-delegee scheme comprises a two-level hierarchy. This argument is also applicable to DOT-based distributed management approaches. Quite felicitously, [MOU98b] characterises this approach as *remote* management rather than *distributed* management. Static hierarchical management approaches address this issue, as management functionality is distributed among a number of mid-level entities and is not concentrated on a single manager platform. Yet, static approaches imply a rigid configuration, which leaves a lot to be desired in terms of management flexibility.

A final point relates to the delegation *trigger mode* supported by both static and dynamic distributed management. In particular, existing approaches address this issue only for the user-driven case. That is, there is no support for delegation as a result of events. In Chapter 6, we investigate how MAs can be used to delegate NSM functionality on an event-driven basis.

[11] In the management context, granularity can be defined as the relative scale, detail, level of hierarchy or depth of penetration that characterises management distribution.

2.13. Summary

Existing management distribution approaches offer a number of mechanisms for delegating NSM functionality to selected entities that perform management tasks as close to the managed systems as possible. As discussed in the preceding section, although the mechanisms have been defined, problems related to the delegation aspects have not been fully addressed yet. Mechanisms only represent a piece in the puzzle of distributed management. There are still open questions on what to delegate, where to delegate and under which circumstances and conditions to delegate.

The ideas behind MbD will however comprise the basis and enabling mechanism for distribution. These ideas will be driven much further by exploiting the power of the MA paradigm that promises to enhance the flexibility of the management process, whilst maintaining the required degree of distribution and scalability. As elucidated in the previous section, MAs can be used to accomplish a synthesis of the strengths identified in static hierarchical models and dynamic management approaches, serving both as a means of functionality delegation and dynamically deployed mid-level management entities. The following chapter discusses the issues related to the application of MA technology on NSM.

CHAPTER 3

MOBILE AGENTS AND THEIR APPLICATIONS ON NETWORK MANAGEMENT

3.1. INTRODUCTION

Code mobility is not a new concept. In the recent past, several mechanisms have been designed and implemented in order to move code among the nodes of a network (e.g. remote batch job submission [BOG73]). A more structured approach has been followed in distributed operating systems research. New approaches to application design have been developed embodying the notion of code mobility, i.e., *the capability of dynamically moving the components of a distributed application among the nodes of a computer network* [CAR97]. The key idea behind code mobility is to provide an alternative to the traditional Client/Server (CS) structure of distributed applications, thus enabling a better use of bandwidth resources and a higher degree of flexibility and reconfigurability. In particular, mobile code-based approaches give rise to significant network load savings through replacing remote CS network communication by local interactions (see Figure 3.1).

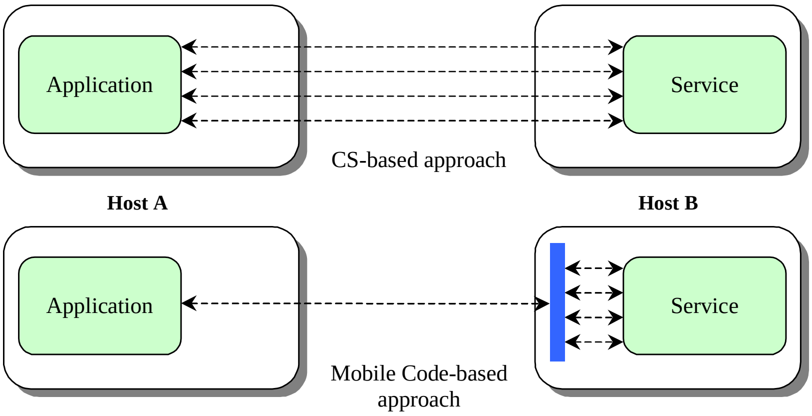


Figure 3.1. CS vs. Mobile Code-based approaches

Furthermore, embedding *migration* capabilities into the mobile code provides an extra level of autonomy and brings forth the concept of the *Mobile Agent* (MA). In general, a *software agent* can be defined as a computational entity, which acts on behalf of others, is autonomous, proactive and reactive, and exhibits capabilities to learn and co-operate [NWA96]. Software agents can be classified as *stationary* (static) or *mobile*. An MA is a software agent with the ability of autonomously moving form host to host.

Given the definition of mobile code, it is clear that this paradigm can provide the technology needed to achieve management distribution. Mobile code can be linked dynamically on network devices either proactively by the manager or reactively by the network device [FUG98]. This way, the management primitives embedded within mobile code become available on the device only when requested by management operations, thus consuming device resources only when this is really needed. Furthermore, MAs add a new dimension in distributed management due to their ability to autonomously migrate to different network devices and perform complex management tasks without the manager's intervention [PIC98].

As a result, mobile code and MAs in particular represent a hot trend in distributed management arena, reflected to numerous research activities [CHE98, BIE98b]. Similarly to other application domains, a lot of hype surrounds MAs in network management and has given rise to high expectations. Yet, until present, there has been very limited interest from industry to support MA-based management, and there are no clear signs of a near-term take-off of this technology. This is mainly due to several open issues still not sufficiently addressed by the agent community (e.g. security concerns), while it is still unclear whether the use of MAs in management applications may actually offer performance gain. This chapter first introduces the reader to mobile code and MA technologies, surveys research initiatives in MA-based management and attempts to identify suitable models that enable the effective use of this technology in the management domain.

The remainder of this chapter is organised as follows: Section 3.2 provides an overview of code mobility paradigms, with Section 3.3 listing their advantages when used in distributed applications. Section 3.4 highlights the differences between MAs and the other mobile code paradigms, suggesting application scenarios where the former can offer performance or flexibility benefits over he latter. Section 3.5 focuses on agent mobility describing aspects that affect the design of MA platforms. It also reviews mobile agent languages and standardisation approaches, classifies mobility schemes, briefly describes commercial Mobile Agent Platforms (MAP), lists application fields utilising MAs and discusses performance issues related to their practical use. Section 3.6 concentrates on MA-based management: it describes several MAPs tailored to Network & Systems Management (NSM), provides a survey of research activity on the field, discusses performance aspects related to these applications, and suggests ways to effectively utilise MAs in terms of organisation and mobility models to optimise the performance of management applications. Finally, Section 3.7 summarises the chapter.

3.2. CODE MOBILITY PARADIGMS

Mobile code paradigms encompass different technologies, all sharing a single idea: to enhance flexibility by dynamically transferring programs to distributed devices and have these programs executed by the devices. The program transfer and execution can be triggered by the device itself, or by an external entity. Fuggetta et al. [FUG98] made a detailed review of mobile code, where they clearly define the boundaries between technologies, paradigms and applications. In particular, they identified three different types of mobile code paradigms: (a) Remote Evaluation, (b) Code on Demand, and (c) Mobile Agents (see Table 3.1).

	Initiator (A)	Co-operator (B)
Client-Server	–	**Code** **Data** **Processor**
Remote Evaluation	Code	**Data** **Processor**
Code on Demand	**Data** **Processor**	Code
Mobile Agent	Code	**Data** **Processor**

Table 3.1. Design paradigms for mobility (the boldfaced typesetting indicates where the interaction takes place).

3.2.1. Remote Evaluation

In the Remote EValuation (REV) paradigm [STA90], a client (initiator) has the know-how necessary to perform a service but it lacks the required resources, which are located at a remote server (co-operator). Consequently, the client sends the service know-how to the remote site that executes the code using the resources available there. This is a form of *push*. The information transferred includes the agent code plus a set of parameters (arguments). After executing the operation, the remote site returns the results back to the initiator of the remote evaluation [ROT97].

Given the above definition of REV, it is clear that several standardisation and research approaches on distributed management, exemplified by the Management by Delegation paradigm (e.g. Script MIB, MbD, flexible agents, spreadsheet approach, etc), can be considered as direct application of REV on the NSM area, as they all share the idea of moving NSM functionality from the managers (initiators) to the management agents (co-operators).

3.2.2. Code On Demand

In the Code On Demand (COD) paradigm, the initiator *A* is already able to access the resources it needs, which are co-located within the same device. However, it lacks the information on how to process such resources. Thus, *A* interacts with its co-operator *B*, requesting the service know-how. A second interaction takes place when *B* delivers the know-how to *A*, which can subsequently execute it. This is a form of *pull* [FUG98].

A widely used technology based on COD are the Java *applets*: applets are programs written in the Java programming language; when a Java-enabled browser is used to view a web page that contains an applet, the applet's code is transferred to the end system and executed by the browser's Java Virtual Machine (JVM).

3.2.3. Mobile Agents

In the MA paradigm, the service know-how is owned by the client, but some of the required resources and data are located at a remote server. Hence, component *A* migrates to the server carrying the know-how and possibly some intermediate results. After its arrival, *A* completes the service using the resources available there [FUG98]. The MA paradigm differs from other mobile code paradigms on that the associated interactions involve the mobility of a computational component. In other words, while in REV and COD the focus is on the transfer of code between components, in the MA paradigm a whole computational component is moved to a remote site, along with its state, the code it needs, and some data required to perform the task. In that sense, MAs can be regarded as a 'superset' of REV/COD paradigms, as they can offer all the functionality provided by the latter, with the additional ability of autonomous migration.

3.3. MOBILE CODE - ADVANTAGES

Mobile code technologies represent a powerful programming paradigm, which is useful when designing distributed applications. The commonly agreed benefits of mobile code have been discussed in many research papers [HAR95, GRE97, BAL97, PIC98] and summarised in the following:

- *Enhanced Flexibility*. Clients typically access the resources hosted by a server through a set of services, whose interface is typically predefined and commonly agreed among the client and the server. Mobile code can be used to extend and update dynamically capabilities of applications, thereby enhancing systems flexibility.

- *Exploitation of increased resources availability*: Managed devices resources are characterised by continuously increasing availability in terms of processing power, disk and memory capacity. Mobile code takes advantage of that feature to achieve processing load distribution. In NSM field, this advantage can be exploited to perform collection and filtering of management data locally, in a distributed fashion. That way, expensive platforms (managers) dedicated to issuing management requests, collecting, analysing and presenting data are not any longer necessary.
- *Reduction of network traffic*: The transfer of mobile code to the source of data creates less traffic than transferring the data, as mobile code can perform *semantic compression* of data, delivering pre-processed, high-level information.
- *Asynchronous interaction*: Once downloaded, mobile code can perform distributed tasks, even if the delegating entity does not remain active.
- *Interaction with real-time systems*: Installing mobile code close to a real-time system may prevent delays caused by network congestion. In NSM, this problem arises when a number of successive interrelated Management Information Base (MIB) values need to be retrieved to present a snapshot of the system's state, e.g. on Simple Network Management Protocol (SNMP) table retrievals [PIC01].
- *Support for heterogeneous environments*: Mobile code is separated from the hosts by an environment able to receive and instantiate the received code. If the framework is in place, mobile code can target any system, especially when the framework is implemented by a platform-independent language, e.g. Java. The cost of running a JVM on a device is decreasing. Java chips will probably dominate in the future, but the underlying technology is also evolving in the direction of ever-smaller footprints (e.g., *picoJava*[1]).

3.4. MAs vs. REV and COD

Since all the mobile code design paradigms allow the dynamic relocation of the components of a distributed application, a legitimate question to ask is whether one should choose the purest MA paradigm or just an approach exploiting COD or REV. Several researchers have tackled the problem of finding out what are the potential assets of MAs (see for instance [HAR95, FUG98, LAN99]). According to [MIL99], the application domains in which MAs have potential deployment are: (a) data-intensive applications where the data are remotely

[1] The picoJava [picoJava] core is a small, flexible microprocessor core that directly executes Java bytecode instructions; picoJava can be used to run applications in small electronic appliances such as organisers, pagers, and cell phones.

located; (b) extensible servers; (c) applications in which agents are launched by an appliance (e.g. a cellular phone) to a remote server, where the user does not necessarily have to stay connected waiting for the MA's return. However, as it has been shown in the preceding section, the use of REV/COD suffices in the first two application domains.

Certainly, every MA-based application could be alternatively designed using existing established technologies [CHE95]. As with every design choice, the answer is given by application requirements and engineering tradeoffs. Detailed comparative analysis of the three mobile code paradigms can be found in [PUL99], with [BAL98] focusing on their quantitative evaluation, using NSM applications as a case study. In addition, [MAG96, BIE98b, CHE98] discuss the potential of MAs in management applications and [HAY99] explores their assets in the wider field of telecommunications. In this section, we identify some aspects related to distributed applications design where MAs can offer more efficient or flexible solutions compared to REV/COD, with special focus given on distributed NSM applications. Given that the distributed NSM approaches reviewed in Sections 2.9 & 2.10 apply the REV paradigm concepts in the NSM area (see Section 3.2.1), the advantages of MAs over REV/COD are, in effect, valid also when comparing MA-based solutions against existing distributed management approaches.

- *New programming paradigm*: MAs provide a subjective advantage because of the metaphor they embody. Agents that are able to dynamically and autonomously relocate themselves according to the application needs may provide, for certain applications, the building block of a uniform and elegant design where every active component is able to spontaneously relocate itself. This characteristic of MAs is probably at the core of their popularity, and provides also a link to other disciplines, like artificial intelligence, that brought this concept to the extreme by proposing *agent-oriented programming* [JEN00] as a new way to create distributed applications.

- *Decreased dependency on a master process*: Approaches based on REV or COD have increased dependency on a master process (delegator) that communicates with one or more mobile code components for controlling and co-ordinating their tasks. Conversely, MAs carry out these tasks in a programmable and autonomous fashion that obviates the need for co-operation with the master process [KAW00]. In this sense, the interaction between the MA and the delegator is limited to the stages of transmission and return of data [PUL99].

- *Space savings*: Resource usage is limited, because an MA resides only on one node at a time. In contrast, static extensible servers require duplication of functionality at every location. MAs carry the functionality with them, so it does not have to be duplicated [BIE98b].

- *Easier update of decentralised tasks*: REV and COD paradigms are not suitable in cases that mobile code needs to be frequently updated, since every update involves broadcasting the updated code to all devices, resulting in excess usage of network resources. In the MA paradigm, such updates are easier and more efficient as the MA code is updated at a central location (code repository) and does not involve network interactions.
- *Traffic around the manager station*: A main priority in NSM is to reduce the traffic around the manager platform [BAL98]. REV and COD imply a pairwise interaction between the manager and each managed device, where mobile code has been installed [PIC01]. Should the manager-managed systems communication is relatively frequent or the size of the managed network is fairly large, the associated traffic will affect the manager's network neighbourhood, although it is very much reduced in comparison with SNMP traffic. On the other hand, MAs once unleashed can visit the devices autonomously, without requiring any communication with the manager until all the results have been collected. Thus, no matter how many devices are visited by the MA, the manager is involved only in the initial dispatching of the agent and in the final collection of results, while the remaining traffic is steered by the MA away from the manager.
- *Management of remote subnets*: Considering the management of remote subnets, connected to the manager site through low-bandwidth links, when using REV or COD the traffic over interconnecting links increases with the number of devices residing in the remote subnets and the frequency of communication required with the manager station, due to the pairwise interaction between the manager and each managed NE. MAs offer a more efficient solution as they need to traverse the interconnecting links only twice (to visit the remote subnets and return the results), regardless of the number of visited devices. Even if the state of the MA increases during this operation, bandwidth is assumed to be 'cheaper' within the LAN than on the low-bandwidth link.
- *Short-term distributed tasks*: When distributed tasks are intended to run over a set of devices for a relatively short period, it is more efficient to use a MA-based approach, where a MA object sequentially visits the devices rather than broadcasting mobile code and obtain the results from every NE. That also reduces the deployment time, especially when the management of multiple NEs is involved (only one MA is issued by the manager) [BOH00b].
- *Reduced deployment cost and delay*: MAs can be used to dynamically increase availability of certain services. For example, the density of fault detecting or repairing agents can be increased upon detecting malfunctions, through creating *clones* of existing MAs and dispatching them to areas of concern. Hence, the traffic and latency involved in

the deployment procedure can be significantly reduced, as it is carried out autonomously and not initiated from a central location [LIO99].

- *Local vs. global semantic compression of data*: By visiting a number of devices, MAs can also extend the concept of semantic compression enabled by REV and COD. These paradigms can provide only a form of compression that is limited to a single device, and local to it. The MA paradigm, in contrast, enables global semantic compression of data across all the network devices visited.
- *Device vs. Domain-level view*: In parallel to the previous argument, approaches based on REV or COD imply a local view of the device where the distributed code statically resides. Should domain-level information incorporating data collected from a set of devices is required, data correlation process should be performed at the manager station [LOP00]. Alternatively, a collaboration scheme between static agents could be applied in order to exchange and correlate information [MOU98a], however that would imply a more complex system design. MAs offer a more simple solution to this problem as they can perform data correlation through sequentially visiting the entire set of devices.

Some of the aforementioned arguments are also valid when comparing the MA paradigm against static approaches based on distributed objects, with the additional advantage of enhanced flexibility and reconfigurability of the former over the latter, as MAs allow the dynamically augment management services at runtime, which is not possible when these services are realised through rigidly configured static objects.

Conversely, in many cases a naive use of MAs may lead quickly to a highly inefficient design. In network monitoring applications for instance, the use of a MA that roams the network and collects information may actually lead to a design that performs worse than the conventional one, especially when large amounts of data are accumulated within the MA state at each host [BAL98]. Yet, this problem can be partially addressed by launching MAs able to perform semantic compression of data, thereby keeping their size practically constant. Increased security concerns is another argument that can be used against MA paradigm, as MAs typically visit a large set of potentially malicious hosts and face the risk of tampering. Concluding, MA technology cannot be considered as panacea in distributed applications design; if the motivation for using MAs is to optimise system performance, e.g., in terms of traffic or latency, attention should be paid to the choice between MAs and alternative mobile code technologies. In certain cases, a synergy of the two approaches may be preferable. In addition, there are several open issues related with MAs that must be addressed by MA community to help the wide spread of this technology:

- *Reducing migration overhead*: the MA code should be as lightweight as possible [PIC01];

- *Migration delays*: MAs have been criticised for being associated with migration delays of the order of seconds or even tens of seconds, depending on the agent configuration and functionality [KNI99, BOH00a];
- *Security*: Network devices should be protected from malicious agents (and agents from malicious machines) [VIG98];
- *Fault tolerance*: agents should be able to survive network and machine failures [NWA96];
- *Performance issues*: what would be the effect of having hundreds, thousands or millions of MAs roaming on a network? [NWA96]
- *MAs composition*: it is essential to ease and automate the composition of service-specialised MAs even by novice users, with no programming experience [MIL99].

3.5. Agent Mobility

The field of MAs and mobile code has lately become a hot research topic covered by many networking and software engineering conferences. As it has been shown throughout the preceding sections, MA technology represents a promising programming paradigm, which can enhance the flexibility and scalability of contemporary NMSs. However, its merits and weaknesses should be carefully evaluated to ensure its effective use in NSM applications. This section focuses on aspects related to agent mobility and MA platforms.

3.5.1. Elements of a Mobile Agent Platform

A basic component of a Mobile Agent Platform (MAP) is the MA Server (MAS), equivalent to an ORB in CORBA, that runs on each host where MAs can execute. The main purpose of the MAS is to provide an efficient execution environment able to receive, instantiate and dispatch agents, serve as an interface between incoming MAs and the underlying system resources and offer a set of services required by the MAs to perform their distributed tasks.

Focusing on MAs, according to the definition given in Section 1.1, they comprise three parts:

- the *code* part which defines the MAs’ functionality;
- the *data* part (*persistent state*), including the values of the variables declared within the MA class;
- the *execution thread* (with an execution stack).

MAs state is dynamically updated as a result of their visits and interaction with distributed servers where information is collected. MAPs that provide *strong mobility* (see Section 3.5.5.1) enable the transfer of all three parts on every MA migration. Most platforms involve the transfer of only the code and state information (*weak mobility*).

An agent migration may be initiated either from the MA itself or the hosting MAS server by invoking a `move` primitive, which allows an MA to move to the next server included into its itinerary[2], through an agent transfer protocol (ATP). At the time that the `move` method is called, the MA's state is saved and transferred through the network. At the destination site, the MA state is recovered and the agent instantiated, typically provided with its own thread of execution. ATPs are used to transfer agents between MASs and can be based on several protocols such as sockets, HTTP, Java RMI, etc. In addition, MAPs also provide agent development and deployment facilities, defined in APIs. A set of classes and interfaces are supplied and should be integrated in the agent code in order to enable mobility.

Besides these basic functions, a MAP may include additional services and facilities. *Fault tolerance* features insure that agents are reliably transferred and are not lost due to a system or network failure. Many available MAPs include *directory* and *location* services [LAZ98]. These services allow the agents to be aware of the existence of other agents and to track their locations. Another important service is *security*, which deals with both the protection of hosts from malicious MAs, and the protection of MAs against malicious hosts. Security mechanisms have to ensure authentication, integrity, confidentiality and access control [FAR96]. Generally, MAPs also provide primitives allowing MAs to *communicate* with each other and with the servers on the visited machines. Inter-agent communication can be enabled by standardised Agent Communication Languages (ACL), such as the Knowledge Query and Manipulation Language (KQML) [KQML] and the FIPA ACL [ACL], developed by the Foundation for Intelligent Physical Agents. These languages have been designed mainly for *intelligent agents*[3]. In addition, a number of proprietary communication models have also been reported in the literature (a detailed description of these models may be found in [INCO]), including direct communication, blackboard, mailbox, meetings and method invocations.

3.5.2. Mobile Agent Languages

MA-based applications can, in principle, be developed in any programming language. However, there are practical issues that render certain kinds of languages potentially more

[2] The term itinerary refers to the list of devices to be visited by the MA. Itineraries may be pre-specified or determined on-the-fly (see Section 3.5.5.3).

[3] The term intelligent agent derives from Distributed Artificial Intelligence (extension of Artificial Intelligence) and refers to a software component involved in a cooperative effort to resolve a problem. Intelligent agents are typically composed of a communication mechanism, a rule base, a solution base and an inference engine [MUL98].

suitable for programming MAs. Given the heterogeneous nature of modern network devices, *portability* is a first requirement (guaranteed by interpreted languages), while additional features that enable *easy development of mobility characteristics* are also of major importance. Other factors include *object orientation, performance*, etc.

Telescript [WHI96] was an early interpreted programming language for MAs developed by General Magic Inc. The Telescript interpreter included a built-in mechanism for transparent migration. Unfortunately, General Magic does not support Telescript anymore. *Agent Tcl* [GRA95] is another MA language developed in the early period of MA technology. Agent Tcl also supports transparent migration and can be useful for running existing Tcl scripts.

The popularity of *Java* has greatly influenced MA-based application developers. As a result, *all* the MA platforms presented in Sections 3.5.6 and 3.6.1 are implemented in Java. In addition to the generic benefits of Java highlighted in Section 2.6.1.1, its suitability for MA programming is enhanced due to its inherent support for dynamic class loading, serialisation, remote cloning and distributed objects communication. These features are discussed below:

Dynamic class loading: Java architecture enables the developer to write programs that dynamically extend themselves by choosing at runtime classes and interfaces to load and use. In fact, some of those classes and interfaces may not even exist when the program is compiled [VEN98]. To enable a Java program to dynamically load classes not included within the local name space, a customised *ClassLoader* (CL) object must be provided to obtain the classes implementations (bytecode) and load them at runtime. A CL is defined by extending the abstract `java.lang.ClassLoader` class and implementing its `loadClass()` method. For instance, customised CLs may dynamically load classes received through the network. This feature makes Java particularly attractive for mobile code-based applications. More sophisticated CLs may even allow to *reload* classes that have been already loaded, in case their implementations have been modified at runtime (see Section 4.4.3).

Serialisation: Java also provides the serialisation feature [OSS97], which allows object instances to be exchanged between different JVMs. Serialisation provides a means for translating a graph of objects into a stream of bytes which can be sent as a message over the network or written in a file. Each instance of a class implementing the `java.io.Serializable` interface is eligible for serialisation. The `writeObject()` method of the `java.io.ObjectOutputStream` class defines the default behaviour for serialising an object, i.e. converting the object's state to a stream of bytes. By default, all the objects referenced by a serialised object are serialised (they must implement the `Serializable` interface); only the fields declared as *transient* or *static* [ARN96] are excluded from the serialisation process. The symmetric process of recreating the object from its serialised representation is termed *de-serialisation*. De-serialisation is achieved by the `readObject()` method of the `java.io.ObjectInputStream` class. The serialisation/ de-serialisation feature of Java is extremely useful when developing MA-based applications. As discussed in Section 3.2.3, a fundamental aspect of agent mobility is the ability of MAs to maintain their *state* information while migrating from one host to another. State can easily be obtained upon migration through serialisation and subsequently recovered on the destination host through de-serialisation. In Appendix B, we propose several ideas for reducing an MA's state size and therefore minimising the migration overhead. Appendix C presents the results of experiments that provide a better understanding of the serialisation process.

Remote Cloning: Java provides inherent support for objects cloning (i.e. creating identical copies of Java objects) through the `clone()` method of the `java.lang.Object` class [ARN96]. Cloning is inspired by the *fork* process mechanism adopted in UNIX. Although support for cloning has not been implemented with mobility in mind, it can greatly benefit MA programming, as it enables the autonomous creation of MA clones at remote sites given that specific conditions are satisfied. Remote cloning can reduce the latency and traffic involved in MAs deployment, thereby enhancing the scalability and flexibility of MAPs [LIO99].

Remote Method Invocation (RMI): Java RMI [RMI] facility provides mechanisms for objects communication in a location-transparent way (see Section 2.5.2). That can be very useful for implementing a communication scheme between agents or agents and other Java applications.

On the other hand, two problems associated with MA programming in Java have been identified:

- It is not possible to implement strong mobility through Java, i.e. to carry MAs execution stack along with their code and state. However, the majority of distributed applications employing MAs can be implemented utilising weak mobility [CAB00].
- Another limitation of Java is that it does not allow the serialisation of classes not implementing the `java.io.Serialisable` interface, e.g. threads (instances of the `java.lang.Thread` class). In Section 6.3.4, we propose a way to get around this problem.

As a result of the aforementioned advantages, we have chosen Java for the development of our MA-based management framework, described in Chapter 4.

3.5.3. Security in Mobile Agent Systems

Security issues related to MA systems are of major concern and have prevented to a large extend the adoption of agent technology by commercial management platforms. Generally, network operators are worried about the capabilities of having self-replicated MAs roaming into communication networks. This behaviour closely mirrors that of a computer virus. A slight change in the agent's executable code is enough to turn an MA to a vicious virus.

The security problem is, in fact, twofold: hosts should be secured from malicious agents [SAN98] and agents from malicious hosts [KAR98b]. The first problem has been more extensively addressed, yet both need to be solved before we can use MAs in real distributed system environments. These security problems have proved harder to solve than people initially expected. As a result, MA security is one of the hottest topics within the agent research community. For a good introduction to the whole range of security issues, the interested reader is referred to [VIG98]. In our MA framework, we have addressed the problem of malicious agents, through a security component that provides authentication, and access control services, whilst offering data encryption to protect sensitive management information from malicious hosts (see Section 4.4.1.3.2).

3.5.4. Standardisation Approaches

The well-known advantages of standards also apply in the agent mobility field, as they allow MAPs inter-operation. In order to establish a common basis for future developments and enable the interoperability of agent platforms developed by different manufacturers, two bodies promote MA standardisation. OMG has defined basic interoperation capabilities between heterogeneous MAPs in its Mobile Agent System Interoperability Facility (MASIF) [MASIF]. MASIF proposes the standardisation of agents, agent system names, agent system types and location syntax; it defines two interfaces (`MAFAgentSystem` and `MAFFinder`) which should be implemented to provide agent management and agent tracking, respectively, functionality. In parallel, FIPA [FIPA] is focusing on the standardisation of basic capabilities of *intelligent agents*. Although the scopes of the two standards are quite different, an interworking or even integration of MASIF and FIPA may be possible in the medium-term time frame [ZHA98a]. Examples of MASIF and FIPA-compliant MAPs will be given in Sections 3.5.6 and 3.6.1.

3.5.5. Taxonomies of Mobility Patterns

This section attempts a classification of the mobility patterns in terms of several characteristics: their support to retaining MAs execution state, their migration strategy (the number of hops realised by MA objects) and their itinerary control (predefined or dynamically configured itineraries).

3.5.5.1. Weak vs. Strong Mobility

Existing mobile code languages provide support for at least one of the following [CAR97]:

- *Strong mobility*: the ability of processes to move their code and execution state to a different site. Processes are suspended, transmitted to the destination site, and resumed there. For instance, Telescript provides mechanisms to implement strong mobility.
- *Weak mobility*: the ability to transfer code across different execution environments; code is accompanied by its persistent state, but no migration of execution state is involved (see Figure 3.2). For instance, Java supports only weak mobility.

Sun's JVM does not allow capturing of processes' execution states and, as a result, very few Java-based MA systems provide strong mobility. Those that do, fall into three categories: systems using a *modified JVM* [ACH97, PEI97], a *custom JVM* [SUR00] and systems using a *pre-processor* [FUN98] approach. Clearly, the implementation of frameworks supporting strong mobility is not a trivial task, whilst introducing performance penalties in agent transfers [FUN98]. In addition, management tasks of configuration, maintenance and control typically involve the execution of repetitive tasks on every node. That means that the requirements of MA-based NSM can be comfortably met by frameworks that only support weak mobility [CAB00, CHE00b]. This statement is also proved by the remarkable precedence of Java over other programming languages that support strong mobility. Our MA platform, described in Chapter 4, supports only weak mobility.

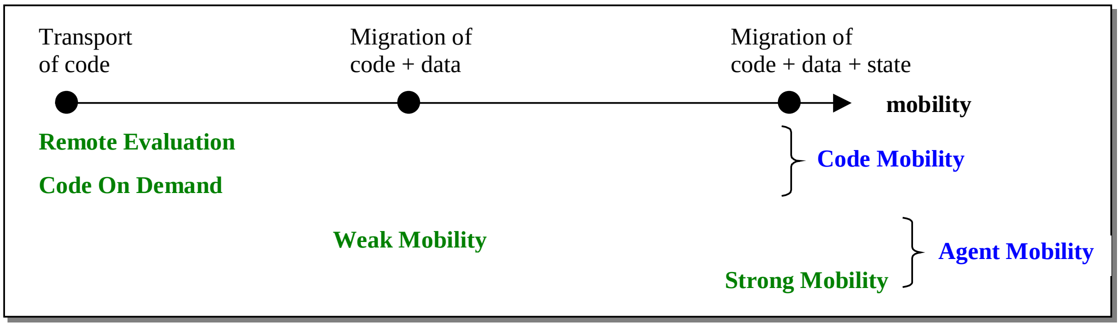


Figure 3.2. Classification of code mobility and agent mobility paradigms

3.5.5.2. Single-hop vs. Multi-Hop Mobile Agents

A second classification of MAs is based upon their migration plan, i.e. on whether MAs visit one or more hosts. *Single-hop agents* travel to a target host, start their execution and remain there until they terminate. This type of agents do not need any data when migrating to the target host (except maybe initialisation data) nor any methods for itinerary control. Therefore, single-hop MAs compare to downloadable code, i.e. they represent a direct application of REV paradigm. Bohoris et al. [BOH00b] use the term *constrained mobility* for single-hop agents.

In contrast, *multi-hop* or *itinerant agents* can travel to several sites during their lifetime. Multi-hop MAs are suitable for performing repetitive tasks over a set of devices. They can also perform different tasks, and can adapt their behaviour depending on the tasks achieved in the previously visited hosts. In [BOH00b], multi-hop MAs are further categorised in *weak* and *strong* MAs, with the former referring to the migration of an MA without preserving information gathered from previous visits and the latter involving the migration of MAs that preserve their state formed during previous visits (standard use of MAs). Weak MAs are termed *memoryless*[4] agents in [CHE00b]. To avoid confusion with the well-established definition of weak and strong mobility given in the preceding section, we adopt the term memoryless (multi-hop) MAs to refer to agents that cannot (can) preserve their persistent state when migrating. As shown in Chapter 5, the choice between single-hop and multi-hop MAs is application-dependent.

3.5.5.3. Itinerary Control

A last classification of MAs is in terms of the control mobile objects maintain on their itinerary. Thus, MAs can either have *fixed* or *dynamic* itinerary [CHE98]. When the itinerary is fixed, the agent migration path is known at the MA's creation time and it does not change during the agent's execution. For instance, fixed itinerary agents are suitable for visiting a pre-determined list of devices to collect data, where the itinerary is typically supplied by the user. In contrast, agents with dynamic itinerary may change their migration path during their execution. The support for dynamic itinerary is achieved though at the expense of increased complexity and size on the MA's code. A discovery agent that is sent to discover new components in a network is an example of agent with dynamic itinerary, since its migration path is specified depending on the detection of new components or subnets. The migration schemes incorporating MAs with fixed and dynamic itineraries are termed *passive* and *active migration*, respectively.

It is noted that migration decisions are either made by the MAs themselves or other entities co-located at the same execution environment. Our MA framework, presented in Chapter 4, supports only fixed itineraries (in performance management applications, the list of monitored devices is known in advance), while migration decisions can be made by both the MAs and the local MASs.

[4] Both single-hop and memoryless mobility involve the transfer of agents code but not their persistent state. However, this is inconsistent with the definition of mobile agents, according to which an MA is a computational entity that carries both code and state information. In other words, the term Mobile Agent is used abusively herein.

3.5.6. Commercial Mobile Agent Platforms

The phenomenal popularity of MAs is reflected on several industrial initiatives that led to the development of numerous MAPs [MAL]. State-of-the-art reports on general-purpose MAPs can be found in [KRA98, COR98b, INCO], with interesting comparative performance, robustness and scalability tests reported in [SIL00, MARINE, MIAMI98]. In this section, we briefly review four representative and popular general-purpose MAPs, all implemented in Java: Aglets, Concordia, Voyager and Grasshopper.

Aglets [Aglets]: The oldest and most well-known platform, developed at the IBM Research Laboratory in Japan. The first version was released in 1996. The migration of Aglets is based on a proprietary Aglets Transfer Protocol. The Aglets Software Development Kit (ASDK) runtime consists of the Aglets server and a visual agent manager, called *Tahiti*. The ASDK provides a modular structure and an easy-to-use API for Aglets programming and also extensive support for security and synchronous/asynchronous agent communication.

Concordia [Concordia]: It has been developed by Mitsubishi Electric. This platform provides a rich set of features, like support for security, reliable transmission of agents, access to legacy applications, inter-agent communication, support for disconnected computing, remote administration and agent debugging.

Voyager [Voyager]: It is probably the most popular MAP, in terms of number of users. It has been developed by ObjectSpace. Voyager is an object request broker with support for MAs. The agent transport and communication is based on a proprietary ORB on top of TCP/IP. Voyager has a comprehensive set of features, including support for agent communication and agent security and also provides support for CORBA and RMI.

Grasshopper [Grasshopper]: Grasshopper has been developed by IKV++, with its main power lying on its compliance with FIPA and MASIF standards. MASs in Grasshopper comprise a *core agency* and a set of one or more *place*s (runtime environments where agents run). The core agency offers a set of services to support agent migration and execution. The communication service that supports agent communication and migration may use a variety of protocols: CORBA IIOP, Java RMI and plain sockets. Grasshopper also offers registration services to keep track of running places and the agents running at each place. Security in Grasshopper allows protecting both MAS-region interactions and agent-MAS interactions. Fault tolerance mechanisms are also integrated, ensuring that agents and MASs can recover in case of crashes or faults [BAU99].

The advantages of using one of the MAPs presented above, are that (a) they are relatively easy to use and typically well-documented, providing attractive frameworks for the rapid development of MA-based distributed applications; (b) most of them are robust, reliable and well-tested. Experiments, however, have demonstrated that these MAPs do not satisfy all performance requirements, as they either involve increased migration latency (Aglets, Voyager, Grasshopper) or poor scalability and robustness under stressing conditions (Concordia) [SIL00]. In fact, Grasshopper has shown to perform worse than others MAPs [MARINE], however its rich functionality, security features and compliance with well-established standards have been the main factors that contributed to its recommendation as the appropriate development platform for several MA-related EU projects [MIAMI, MARINE].

From the management viewpoint though, there are several weaknesses shared between commercial MAPs:

- *Rich, but unnecessary functionality*: Being general-purpose frameworks, most available MAPs incorporate rich functionality, yet, usually unnecessary for management applications. Some of these features are ‘hard-coded’ within an ‘Agent superclass’ that has to be extended in order to implement application-specific MAs and result in large MA sizes that affect the usage of system and network resources. A subset of the provided features would suffice for the majority of management tasks, however the exclusion of the non-desired features is not feasible;
- *Lack of essential features*: Some features considered as essential when designing a flexible management system are not supported by most general-purpose MAPs. For instance, the ability of MASs to distinguish between different versions of the same MA class, which may reflect the update/modification of an existing management task;
- *Heavyweight migration schemes*: The minimisation of MAs migration overhead is of major importance for large-scale monitoring applications that involve frequent polling of a large set of NEs. Existing MAPs incorporate complex and heavyweight migration protocols that result in increased network overhead, which typically exceeds that of static distributed objects communication mechanisms [KNI99, BOH00a];
- *No open-source MAPs*: At the time this research commenced, there was no open-source MAP that would allow the author to modify the code and perform application-specific optimisations;
- *Questionable support*: Companies shipping commercial MAPs, often suspend their support, e.g. General Magic has discontinued the Odyssey project, while that also seems to be the case with IBM’s Aglets [INCA].

All these weaknesses make difficult the selection of a general-purpose commercial MAP as implementation platform and led to the development of numerous application-oriented MAPs, many of which are tailored to management applications (see Section 3.6.1).

3.5.7. Applications of Mobile Agents

MAs can be useful in several application fields, although none of them necessitates their use; in fact, each application can be designed based on existing technologies [HAR95, MAG96]. However, the use of MAs can contribute to build these distributed applications in a more simplified and effective way. In the following, we identify some areas in which MA technology can actually give a positive contribution. NSM applications are not mentioned herein as they will be elaborated later.

- *Information retrieval*: MAs can be an effective tool for retrieving information within a distributed system; in fact, an agent encapsulating the user’s query can migrate to the place(s) where the information is actually stored; therein, the agent can obtain and filter data, and return the user only the useful information [ISM99]. This idea has been used in [PAP99] to reduce the latency involved in remote database interactions.

- *Electronic commerce*: E-commerce is an increasingly expanding area in the Internet; MAs can help users to search the products that meet their requirements, find the most cost-effective offers, etc. [DAS99].

- *Mobile computing*: Users want to access network resources from any position, notwithstanding the band limits of current wireless technologies. Thus, users can submit their requests through an agent, which runs their request within the network and returns the results later (so the user does not need to remain connected, waiting for the results) [CHE95, MIL99].

- *Distributed Computation*: MAs represent new paradigm for parallel execution of computation-demanding tasks on a distributed network of workstations [SIL99b, GHA99].

3.5.8. Performance Evaluation of Mobile Agents

It is often argued that the advantage of agent migration lies in the reduction of (expensive) communication costs by moving the code to the data rather than the data to the code [HAR95]. Although this argument is understandable from an intuitive point of view, not much research has yet been conducted to evaluate the cost of agent migration on a quantitative basis. Performance models regarding network load and execution time are needed to identify situations on which agent migration is advantageous compared to RPCs and help to decide which interaction model to be used.

A performance model for MA systems is introduced in [STR97]. The conclusion drawn from this model is that an alternating sequence of RPCs and agent migrations may perform better than a pure sequence of RPCs or a sequence of agent migrations. In particular, it is argued that agent mobility reduces both the overall latency and network load in cases that MAs visit devices where the amount of data to be processed is large compared to the size of the agent and the *selectivity* of the agent, i.e. its ability to reduce the size of the returned data by remote processing, is high. This conclusion was validated by experimental measurements in a realistic Internet-scale network scenario, using a prototype MA system implementation (Mole [STR96]). Along the same line, Chia et al. proposed *strategic mobility* [CHI97], whereby agents may choose to migrate to selective resources or instead to communicate with the needed resource over the network, depending on the problem's characteristics and the underlying computing and network infrastructure.

NSM-oriented performance evaluations of MA platforms have been reported in several research papers [BAL98, PIC98, RUB99, BOH00a], reviewed in Section 3.6.3.

3.5.9. Discussion

It should have already become evident that MA technology cannot be considered as an ideal solution for structuring *every* distributed application. A major problem that has prevented the wide spread of MA technology is the difficulty to identify 'killer applications' [MIL99, KOT99], i.e. applications that would strongly enforce the use of MA technology and become the driving force towards its further spread, adoption and exploitation.

That is a rather controversial issue which reminds of the object-oriented versus procedural programming debate[5]. Although killer applications probably do not exist, there is a number of application scenarios where MAs may improve the performance or efficiently complement REV/COD paradigms (see Section 3.4). In that sense, MAs should be considered only as another tool in the arsenal of distributed application designers [PIC01]. Concluding, we believe that the decision on whether to use MAs or alternative mobile code technologies should follow an unbiased quantitative and qualitative evaluation and depend on the specific characteristics and requirements of the examined applications.

3.6. MOBILE AGENT-BASED NETWORK MANAGEMENT

The potential of MAs in structuring distributed application has been early recognised by the management community, triggering intense research activity on MA-based distributed management. In the field of Network and Services Management, Magedanz [MAG96] was the

[5] At the time that object-oriented (O-O) programming was at its infancy, several developers often claimed that the O-O approach did not have a killer application either, still, it is now well accepted and widely used. While this claim is questionable, it is true that O-O applications can also be implemented in more traditional ways (procedural programming).

first to signal the potential of MAs, describing several scenarios in which their use can offer important benefits. MAs can encapsulate management scripts and be dispatched on-demand where needed. An MA can be sent to a network domain and travel among its elements collecting management data, and return with the data filtered and processed. Sending an MA for this task is a substitute to performing low-level monitoring operations and processing them centrally. Through semantic compression of collected data, the agent size can remain small and save bandwidth usage.

In an interesting study on code mobility, Baldi et al. [BAL97] presented another advantage of using MAs; when the network administrator is connected via an unreliable, costly or lossy link, he/she can create MAs off-line, connect to the network to dispatch the agents, close the connection and then reconnect later to get his agent back with the results. This principle is actually implemented in Astrolog [SAH97].

This section starts with a description of several MAPs tailored to NSM, overviews several MA-based management applications classified in terms of the mobility scheme they utilise, discusses performance aspects related to these applications, surveys the research activity on the field of active networks which is relevant to the scope of this thesis, and finally discusses efficient organisation and mobility models for MA-based NSM frameworks.

3.6.1. Mobile Agent Frameworks for Network Management

The limitations of general-purpose MAPs, as highlighted in Section 3.5.6, have led to the development of numerous management-oriented MAPs, aiming at optimising flexibility and performance aspects. Simply viewed, these MAPs consist of two components: The MAs themselves and the nodes where they can migrate and execute. There are several names synonymous to nodes where MAs execute, such as Mobile Code Daemon [SUS98], agency [SIL99a], place [ZAP97, BEL99], *lieu* [SAH97], etc. We adopt the term Mobile Agent Server (MAS) for the remainder of this section. In the following sections, we present the most representative platforms explicitly tailored to NSM applications.

3.6.1.1. IMA

The Intelligent Mobile Agents (IMA) framework [KU97], developed in Arizona State University (USA), has been the first MAP intended for building decentralised NSM applications. As such, IMA provides minimal functionality and poor integration with management standards. IMA consists of three major components: the *MA launcher* or managing entity, the *MA code* and the *agent host*. The managing entity is responsible for launching the MAs and processing the data collected by them. The MA is a software program, which migrates among managed entities to collect information based on the policies defined by the managing entity. The agent host is capable of receiving MAs and providing them access to local resources. The agent host runs as a daemon process at each managed entity able to receive and authenticate MAs.

3.6.1.2. MCT

The Mobile Code Toolkit (MCT) has been developed as part of the Perpetuum Mobile Procura project [PMP] in a pursuit of the ultimate goal, a plug-and-play network [BIE97], offering the basic functionality required to deploy NSM MAs. Every node to which MAs can migrate has to run a Mobile Code Daemon (MCD) that includes a Migration Facility and a Mobile Code Manager. The Migration Facility provides transport facilities to the agents. The Mobile Code Manager manages the lifecycle of the agents present on the MCD. The access to managed resources is handled by Virtual Managed Components, which provide a uniform interface for the MAs to monitor and control the visited NE [SUS98]. In [WHI99], an MA injection client is also introduced in order to create, deploy and manage MAs in the network. Also, authentication and data integrity features have been incorporated in the MCT framework.

Several kinds of MAs are defined [BIE98a]. Applets, servlets and extlets are single-hop downloadable components where applets are used for COD and servlets and extlets are used for REV. Deglets are multi-hop agents with limited persistence, that terminate as soon as their task is completed. Finally, netlets are persistent multi-hop MAs.

3.6.1.3. INCA

INCA (Intelligent Network Control Architecture) [NIC98] is an open architecture for the distributed management of multi-service networks. It supports three code transfer schemes: *Push code distribution* is used when the itinerary of the agent is known at creation time, where the code of the agent can be pushed to the end nodes before the agent is launched. In the *pull code distribution*, a station that receives a new MA has first to fetch its code. Finally, in the *migration code distribution*, the code travels along with the MA state. Another original feature in INCA is that the network administrator can assign priorities to MAs, depending on the urgency of their task. Other interesting features include reliable and fault-tolerant communications between stations, monitoring of the agent population deployed in the network and facilities to launch and control MAs execution. In addition, INCA includes a *location service* and a *naming servic*e, used for inter-agent communication. It is noted though, that INCA architecture has *not* been implemented.

3.6.1.4. MAGENTA

MAGENTA (Mobile AGENT environment for distributed Applications) [SAH97], developed in the Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA, France), targets mobile user applications where users are connected to the network with unreliable and expensive connections. Hence, MAGENTA assists the administrators to remotely control their managed network (possibly through portable computers), through launching MAs to carry out distributed management tasks. Places where MAs can travel (MASs) are called *lieu*s. The heavier part of the management functionality is integrated into a static *management server*. The framework is enriched with features such as fault tolerance, adaptability of MAs to changes in the environment (they are capable of detecting the disappearance of a lieu), etc [SAH98]. MAGENTA also offers a limited support of security based on access control.

3.6.1.5. AMETAS

The Asynchronous MEssage Transfer Agent System [AMETAS] developed in University of Frankfurt (Germany), uses the concept of a *place* as a MAS. Each place offers a mailbox system that allows agents to communicate asynchronously. A place can extend its capabilities by installing services that can be accessed through a control interface. MAs can therefore access managed resources after installing appropriate services. Another feature of AMETAS is that it defines human-agent interfaces through the definition of *user adapter*s. Finally, AMETAS uses a complex security mechanism including authorisation, access control and encryption [ZAP97].

3.6.1.6. SOMA

The Secure and Open Mobile Agent (SOMA) architecture [SOMA], developed in the University of Bologna (Italy), concentrates on security and interoperability as its two main design objectives. SOMA has been developed on the top of an MA environment used for NSM, MAMAS (Mobile Agents for the Management of Applications and Systems) [COR98a]. It enforces a strict security model with authorisation, authentication, integrity and secrecy features. Interoperability with other MAPs is achieved through the support of the OMG MASIF specifications [MASIF]. Moreover, a software add-on ensures interoperability with CORBA-compliant distributed applications and allows inter-operation with legacy systems through CORBA gateways [BEL99]. Several abstractions are defined within the MAMAS architecture: The *place* abstraction where agents can execute (i.e. the MAS); the *domain* abstraction which encloses a set of places; domains typically represent LANs and include a default place that embeds the *gateway* abstraction responsible for interconnecting different domains.

3.6.1.7. JAMES

JAMES [JAMES] is a MAP designed for the management of telecommunications networks, developed as part of a collaborative project between University of Coimbra (Portugal) and Siemens. MASs in JAMES are called *agencies*. The *JAMES manager* allows the network administrator to control active agents and agencies. A *code server* provides a central repository where MA codes are stored [SIL99a]. Moreover, JAMES agents have a passive migration strategy according to which the itinerary of each agent is known at launch time. Agencies make use of this property to pre-fetch the agent code from the code server [SOA99]. This leads to improving MA migration performance. JAMES also uses a checkpoint mechanism to provide fault-tolerance. Agencies states are periodically saved on a persistent medium so that they can be recovered after a crash and restarted from the last saved checkpoint. Moreover, MAs are saved and maintained until they reach the next checkpoint.

3.6.1.8. MAP

The Mobile Agent Platform [MAP] has been developed as part of a collaborative project in Universities of Catania and Messina (Italy). MAP architecture includes the following components [PUL00a]: the *Server* (MAS), able to accept and activate MAs; the *Daemon*, listening on a specific port for visiting MAs; the *Context*, that maintains a list of locally executing MAs and manages inter-agent communication; the *NetworkClassLoader* that enables MAs to run on a Server, even when their class is not present therein; the *CodeServer*, integrated within the Context, which stores the classes available at the Server. Additional features allow for synchronous/asynchronous inter-agent communication, to remotely retrieve information about MAs and change their execution state, etc. A key feature of this platform is its compliance with the MASIF standard, ensuring interoperability with other MAPs.

3.6.1.9. CodeShell

CodeShell [BOH00c], developed in University of Surrey (UK), is an optimised mobile code platform supporting the constrained mobility paradigm (see Section 3.5.5.2). As such, CodeShell cannot strictly be classified as MAP, as it does not provide migration facilities to launch MAs. The motivation that led to the development of this platform has been to address the performance limitations (mainly large migration delays) of general-purpose MAPs (see [BOH00a]). The basic components of the CodeShell architecture are a *communication service* (using Java-RMI) which provides a mechanism for delegating management logic along with initial parameters to remote machines and a *naming service* that distinguishes between objects and also binds one object to another.

3.6.1.10. Discussion

In this section, we described a number of MAPs designed and implemented for NSM applications, which address some of the limitations of general-purpose platforms, discussed in Section 3.5.6. All the presented MAPs support multi-hop agent applications, with the exception of CodeShell, which exclusively supports constrained mobility, however none supports memoryless or strong mobility. Interestingly, only SOMA and MAP comply with the OMG MASIF standard (see Section 3.5.4), while Java is the implementation platform in all cases. However, a number of limitations have been identified on these MAPs:

- *Heavyweight migration scheme*: With the exception of few (e.g. [PUL00a]), existing MAPs involve the transfer of both state *and* code at each MA migration. The transfer of code though is unnecessary, unless the MA visits a device for a first time, as the Java CL stores every loaded class on a local code table. That inefficient scheme may result in serious scalability problems both in terms of latency and migration overhead. This problem is partially addressed through the migration strategy proposed in [SOA99] and [PUL00a], where *only* the MA state is transferred and should the corresponding code is not present at a visited device, the device's CL contacts and downloads the code from a remote code server. This approach is very efficient in terms of network traffic (MA code is transferred only when necessary), however it increases the latency (the MA’s execution cannot start until its code is downloaded). In addition, it involves more complex migration mechanisms, which are not necessary when MAs itinerary is known in advance. Instead, we have chosen to adopt the ‘push’ scheme (defined in [NIC98]), whereby bytecode is distributed at the MA’s construction time with only the persistent state transferred thereafter, resulting in minimal usage of network resources (bytecode size is typically much larger than state size [BAL98]) and faster class loading. This migration scheme is described in Section 4.4.1.4.1 with a more refined design detailed in Section 6.3.5.
- *MA services customisation*: The development and customisation of MA-enabled NSM tasks is not effortless with available MAPs, as it requires programming skills and detailed knowledge of the MAPs’ design. In Section 4.4.1.4, we introduce a tool that automates the generation of service-oriented MAs in a user-friendly manner, according to specified operational requirements.
- *Class loading*: Most MAPs include a CL component, able to receive and load at runtime visiting MAs bytecode. Yet, to the best of our knowledge, there is not any MAP which

allows to modify (overwrite) the bytecode, i.e. to dynamically upgrade MA-enabled management tasks. This problem is related to a limitation of Java class loading mechanism, which we address through a customised CL, described in Section 4.4.3.

- *Security*: Not all management-oriented MAPs sufficiently address security issues related to MAs. Examples of platforms without any support of security mechanisms or adopting weak security schemes, include IMA, INCA, JAMES, CodeShell. In Section 4.4.1.3.2, we describe a security component, which is integrated within our MAP and provides authentication, authorisation and encryption services.
- *Fault tolerance*: MAPs should be able to survive situations where link or node failures disrupt the normal migration process of roaming MA objects or the communication with the manager station. However, only few platforms (MAGENTA and JAMES) have addressed fault tolerance issues. In Sections 4.4.2 and 6.3.9, we discuss how network or node failures are dealt within our platform.
- *MA organisation models*: The reviewed MAPs define architectures comprising two hierarchical levels, corresponding to the manager and the NE ends, with the MAs used to delegate NSM functionality from the manager to the NEs. This organisation model is suitable for the management of small or medium-sized LANs but not adequate for large-scale, geographically dispersed enterprise networks, typically structured in logical hierarchies. This problem is discussed in Section 3.6.5.1.

3.6.2. Mobile Agent Applications in Network Management

MA-based distributed management has been a very hot research topic in the past few years, with a large number of proposed applications reported in the literature. This section provides an overview of MA-based approaches in a broad spectrum of applications, including network, systems, fault, configuration and service management. In order to provide a more structured overview, we follow the classification of Section 3.5.5.2, grouping these applications according to the mobility scheme used (single vs. multi-hop agent applications) and the control on agents' itinerary (passive vs. active migration). Most of the applications described below use the MAPs described in the preceding section as underlying platforms, whereas others have chosen general-purpose platforms such as Voyager [FER01] or Aglets [PIN99, CHI99].

3.6.2.1. Single-hop Agents in Network Management

As a general rule, single-hop migration is useful to encapsulate a function or a service into an MA and to deploy it to a remote location. A number of applications make use of this idea. White et al. [WHI99] suggest that a Virtual Managed Component (VMC) resides at each NE, providing incoming MAs an interface to managed resources. The manager side of the NMS requires a similar interface called the Virtual Managed Resource (VMR), which can be viewed as a remote wrapper of the VMC. The VMR can be supplied as a single-hop MA that travels to the remote NE, thereby, seamlessly enabling the management of newly installed NEs. In addition, this allows network component suppliers to transparently use any NSM protocol (even a proprietary protocol) for the management of NEs.

In [PUL00b], four types of MAs (programmed on the top of MAP platform, described in Section 3.6.1.8) are identified to perform management functions. Among them, the *daemon agent* is a single-hop agent sent to a network node to locally compute a health function (a linear aggregation function of several MIB values) and automatically notify the manager station when certain thresholds are exceeded. This scenario provides the advantage of saving bandwidth when compared to a remote polling-based scenario. Single-hop agents are also supported by the CodeShell platform (see Section 3.6.1.9), mainly applied to performance management applications: the aim is to provide traffic rates, QoS alarms and periodic summarisation reports by observing raw information such as traffic counters on NEs [BOH00c].

Other possibilities of applying single-hop agents are suggested in [MAG96]. A service or a network provider can send single-hop MAs to user end-points in order to adapt his equipment to new services. These agents can also achieve other tasks such as user accounting and capturing user requirements.

3.6.2.2. Multi-hop Agents in Network Management

Fixed Itinerary (Passive Migration)

The most typical and widely-used scenario of applying multi-hop agents is to deploy an agent to a list of hosts to locally perform management tasks and return to the manager processed management information. In this context, mobility allows the agent to perform semantic compression [BAL98], take decisions based on the past visited nodes and bring back a report result to the management station. This scenario is proposed in many works for different management activities.

Several research works on MA-based network monitoring involved rather simplistic applications, whereby a single MA object sequentially visits a predefined set of hosts, collecting a number of MIB values from each one without performing any processing upon them. This approach, used in [KU97, SAH98, CHI99] fails to solve the scalability problems of SNMP management as the state of travelling MAs grows rapidly (due to the unprocessed data accumulated into the MA state) resulting in increased network overhead and response time, in addition to imposing computational burden to the manager station where data processing takes place [BAL98]. The same principle is used in [COR98a], where MAs return system resources utilisation reports from a group of devices. Simulation results have shown that using MAs as management data collectors can be more efficient than SNMP-based management only when the management of remote domains (separated from the manager station by bottleneck links) is considered [RUB99].

The ability of MAs to return high-level information is exploited in [PUL00b], which defines a *verifier agent* that returns a list of nodes verifying a certain condition, e.g. overloaded CPU. In [SIL99a], MAs are used in a TMN environment to collect and process performance data from a set of NEs in order to produce *global* reports about the performance of the network. More advanced applications are proposed in [ZAP99] that presents NetDoctor, an application built on top of the AMETAS platform (see Section 3.6.1.5). NetDoctor addresses scalability issues by delegating NSM tasks to MAs that migrate to remote domains where they act as local managers, performing SNMP operations. Several interesting applications are proposed, including evaluation of health functions, termination of mis-behaving processes to free-up system resources, etc. Similar applications have been proposed by Pinheiro et al. [PIN99] that defined *discovery agents* to discover the existence of specific MIB variables on given hosts, and *aggregator agents* to perform computations on MIB variables (several aggregation levels may be defined). An interesting aspect is the dynamic adaptation of the proposed architecture to changing network conditions, so that MAs move closer management data to minimise their intrusiveness, in terms of the NSM-related traffic. This application has been developed using Aglets as underlying mobility framework.

El-Darieby et al. proposed a fault management application, using intelligent MAs endowed with a rule-based engine that allows to infer and diagnose possible faults on visited nodes [ELD99]. Another interesting application is described in [KNI99] that uses MAs for monitoring the conformance of Service Level Agreements (SLA) in enterprise networks. The paper suggests to use MAs that periodically roam the network to collect SLA monitoring results produced by other agents standing close to locations where user applications execute. Both the applications described in [ELD99] and [KNI99] are developed on the top of the MCT platform, described in Section 3.6.1.2.

Feridun et al. [FER01] presented the Distributed Management Framework (DMF), an architecture built on the top of the Voyager platform. The execution environments for incoming MAs are provided by the Distributed Management Nodes (DMN), characterised by a highly modular and lightweight design. The DMF has been tested on an application scenario where an *enterprise manager*, running on a DMN, analyses IP traffic characteristics of remote subnets. Upon receiving an event reporting that a performance threshold has been exceeded, the enterprise manager dispatches an MA to the remote subnet where the event originated from. On arrival, the MA starts monitoring the traffic activity on the subnet using a packet sniffer. When the specified monitoring period ends, the MA returns back to the enterprise manager to deliver its collected data.

The *Mobile Disman* architecture described in [OLI99] goes one step further, integrating the IETF's Distributed Management (Disman) framework [Disman] with a MA-based NSM framework. Disman defines an architecture where a main manager can delegate control above several distributed managers (DM), thereby improving the scalability, robustness and flexibility of centralised approaches. In Mobile Disman architecture, MAs can by used to implement DMs; providing mobility support to DMs allows them to adapt to dynamic environment conditions, offers location transparency and simplifies tasks such as data correlation and tasks distribution. However, Mobile Disman architecture is heavyweight, i.e. it comprises many resource-demanding components that would certainly increase the requirements on system resources; the fact that Disman is still only an Internet draft should also be considered. In addition, Mobile Disman architecture is *not* supplemented by a prototype implementation.

Dynamic Itinerary (Active Migration)

In their pioneer work, Appleby and Steward demonstrated the feasibility of employing multi-hop actively migrating MAs to control traffic congestion in circuit-switched networks [APP94]. A first class of MAs, called *parent agent*s, randomly navigate among the network nodes and collect utilisation information. By keeping track of this information, they gather an approximate utilisation average of the network nodes. Therefore, they are able to identify congested nodes, relatively to this average. When a congested node is found, a *load-balancing* MA is created to update the routing tables of the neighbouring nodes so as to reduce the traffic routed through the congested node. This application inspired other researchers to further develop its ideas by using *biologically* inspired agents. The work of Minar et al. [MIN99] covers this ground, using MAs to configure routing tables in a highly-dynamic radio frequency network where nodes are low-power transceivers, moving from one location to another in a two-dimensional space.

Shoonderwoerd et al. [SCH97] proposed to use *ant*-like MAs to achieve load balancing in telecommunications networks. MAs randomly roam the network and put *pheromones* depending on the distance from the source and the congestion of the followed route. Deploying a sufficiently large number of such ant-like agents allows to route calls according to the distribution of pheromones. Simulations show that such MAs significantly decrease call rejections compared to other approaches. An improvement to this work is proposed in

[BON98], where MAs no longer roam the network in a completely random manner, but follow those places where the strength of pheromones is higher.

White et al. [WHI98] applied a similar approach in a network fault location application, using multiple interacting *swarms* of MAs. Four types of agents are defined. *Service monitoring agents* monitor the compliance of service instances to the required QoS. When significant changes are detected, a *service change agent* is sent to mark the resources on which the service depends (increase the pheromone intensity when the QoS has downgraded). *Condition sensor agents* continuously roam the network and evaluate specific conditions on the visited nodes, with the tendency to visit more frequently those nodes where problems have been detected. Nodes with strong pheromone attract a fourth type of agents called *problem identification agent*s, with the capability to diagnose and repair certain patterns of problems. As a general comment, it should be stated that the applicability of ant-based solutions in the field of telecommunications is questionable; so far it has only been demonstrated in simulated environments.

Active migration has been also applied in network discovery and dynamic configuration of networks and services. In [SCH98] and [WHI99], the authors proposed a scenario, where MAs (*netlets*) continuously roam the network to discover new devices and detect removed components. Netlets are suitable for dynamic networks with frequently changing topologies and supported services. An MA-based network discovery application is also described in [FER01].

In another work, Pagurek et al. [PAG98] developed a simulation testbed for the configuration of Permanent Virtual Channels (PVC) in heterogeneous ATM networks. The idea is to dispatch an MA that travels across the network switches, and progressively configures the PVC fragments on each switch. After configuring the PVC route on a switch, the MA travels to the next switch where it uses configuration information of the past switches to correctly configure the current one. Cheikhrouhou et al. proposed a similar application [CHE00a], based on static intelligent agents with dynamically extended capabilities.

It is essential to notice that in the above applications, MAs provided a very interesting tool to model either lightweight moving entities or biologically inspired entities used for management tasks. The MA metaphor allows to model a self-contained entity that evolves in its environment to accomplish simple tasks according to its *own view* of the network.

3.6.3. Performance of Mobile Agents in Network Management

While the main focus of MA-related research activity on distributed management has been on developing NSM-oriented frameworks and using them on specific applications, some researchers concentrated on evaluating the performance of MA-based management. The purpose of such performance studies is twofold. First, they suggest ways to efficiently use MAs so as to outperform centralised NSM approaches. Second, they allow to determine how MAs are best deployed for particular types of NSM operations.

Pioneer work on this direction has been presented in [BAL97] and [PIC98]. A management task carried out on a set of NEs, involving multiple CS interactions with each NE has been examined. Models of the overall traffic and the traffic around the management station are computed for CS, CoD, REV and MAs approaches. The impact of semantic compression is also studied for mobile code approaches. The purpose of these models is to provide the network administrator with an objective quantitative criterion to choose the right approach, given the management task and the managed network topology.

Liotta et al. [LIO99] presented an evaluation of MA-based monitoring systems, providing quantifiable metrics based on performance and scalability. MAs are used to perform monitoring tasks in place of traditional centralised polling. Performance is measured using the generated monitoring traffic and the monitoring delay. These parameters are evaluated for a combination of schemes based on monitoring models, MAs organisation and MA deployment patterns. Possible monitoring models consider MAs performing periodic polling. In the first model, each MA polls and analyses a set of managed objects (MO); this MA is subsequently polled by other MAs or the monitoring station. In the second model, each MA periodically polls the MOs, analyses the results, and *notifies* other MAs or the monitoring station. The third model differs from the previous on that MAs generate data (alarms) only if specific *events* are detected. The MA organisation can either be flat or hierarchical, and the deployment scheme can either be cloning-based or without cloning. This leads to four possible deployment patterns: flat broadcast with no cloning, flat broadcast with cloning, hierarchical broadcast with no cloning and hierarchical broadcast with cloning.

In a more recent work [LIO01], the authors investigated the problem of optimal placement of MAs acting as remote monitoring stations, so as to minimise data collection latency and the network traffic incurred due to (localised) polling. To address this problem, a distributed algorithm is proposed that relies on agents learning about the network topology through standard management interfaces and subsequent deployment of MAs to remote domains through a ‘clone and send’ process. Deployed MAs can possibly adapt to network changes and move again in order to maintain optimality.

The latency aspects of MA migrations, with respect to management applications, have been investigated by Lipperts [LIP00]. The results of time measurements indicate that MA migrations can be more time efficient than remote communications, especially when class loading is not necessary (the code of the MA is already at the destination host) and the operations are performed over slow media. However, the decision concerning the replacement of a remote communication scheme by MA migrations should take into account several parameters, such as the number of remote communications to be replaced, the size of the parameters involved in the remote communication, the size of the agent and the networking environment. Lipperts proposed a solution based on utility theory to aid on deciding whether MAs deployment is justified or not.

[ELD99] and [ZAP99] included brief quantitative evaluations of their proposed applications demonstrating improved scalability compared to SNMP management. Similarly, [BOH00c] compared the performance of CodeShell against that of Grasshopper, RMI and CORBA-based approaches in terms of response time, generated traffic and memory requirements. The experimental results indicated that CodeShell offers the management flexibility benefits of general-purpose MAPs, whilst achieving performance characteristics comparable to static DOT-based approaches.

3.6.4. Active Networks

Active Networks (AN), which originated at MIT’s Software Devices and Systems Group, advocate the use of *active packets* or *capsules* [TEN96]. Active packets contain user data and programs, which are executed on every node along the route of the packet through the AN. Such packets require a software layer on top of the hardware and communication protocols, which can provide dynamic configuration, improved security, interoperability, extendable protocols, etc. AN research is relevant to the context of this thesis, as programs carried by active packets, resemble the functionality of MAs: an active packet can be thought of as a multi-hop MA, which executes on every node along its path through the network; likewise, a travelling MA can be mapped to an active packet, containing both data and a program, sent from one active node to another, in an AN. Since the motivations for proposing the MA and AN concepts are on the same direction, MA and AN technologies start to converge and overlap, specifically in the management domain [KAW00]. Nevertheless, only recently have AN concepts been applied to distributed management [GRE99, RAZ00, KAW00].

Greenwood and Gavalas [GRE99] described a modular framework that exploits the ability of active mobile processes or agents to interact with one another in order to realise active network architectures. The process execution environment is integrated into the network devices. The process kernel houses both pre-loaded and visiting agent processes. The proposed framework is compared against traditional SNMP-based with the former outperforming the latter both in terms of response time and network overhead when the calculation of health functions combining large numbers of MIB objects is considered.

Raz et al. [RAZ00] described a prototype system where legacy routers are enhanced with an *active engine*, which enables the rapid deployment of new distributed management applications. In particular, the active engine comprises an environment in which code encapsulated in active packets can be executed. Physically, the routers forwarding mechanism and the active engine may either reside on different machines or co-reside in the same box. This prototype has been tested on a simple bottleneck detection application and offers the functionality of the well-known *traceroute* program, in a more efficient way.

The convergence of MAs and ANs is more clearly portrayed in [KAW00] that introduces the Active Distributed Management architecture. This architecture is characterised by a programmable middleware platform (organised in different layers of abstraction), whose active properties are drawn from the AN and MA paradigms. A number of applications are proposed including variables monitor/control, bottleneck detection, topology detection, etc.

3.6.5. Synthesis and Discussion on Mobile Agents-Based Management Applications

This section attempts to highlight the benefits and shortcomings related to the application of MA technology on NSM. The discussion is twofold: first, the effect of MA organisation models is investigated with respect to the scalability and flexibility of MA-based approaches on the management of large-scale networks; second, the effect of mobility schemes on management applications is discussed and ways to effectively exploit the mobility feature of MAs are suggested.

3.6.5.1. Organisation Models

As mentioned in Section 3.6.1.10, a common attributed shared between the MAPs developed for management applications is their structuring in two levels, corresponding to the manager and the managed devices, with the MAs used to delegate functionality from the former to the later. This simple organisation model implies a ‘flat’ network topology where a single MA is launched and visits the entire set of managed systems, causing scalability problems both in terms of latency and network overhead. Flat models are not suitable for the management of large-scale, hierarchically structured networks, which can be more efficiently managed by MA platforms organised in hierarchical fashion [LIO01].

The hierarchical organisation of agents is not an entirely new idea. For instance, hierarchies of *static* co-operating agents have been proposed in [QUE97] to support the resource-control part of a signalling system. In the management domain, this problem is partially addressed by the hierarchical framework described in [LIO98] and also MAGENTA, AMETAS and Mobile Disman architectures. In particular, Liotta et al. [LIO98] proposed an MA-based management architecture adopting a multi-level approach enabled by static *Middle Managers* able to launch MAs. The same principle is used in MAGENTA platform. A second approach has been reported in [ZAP99] and [OLI99] that proposed the use of MAs as mobile mid-level managers, performing distributed management tasks. The first approach [SAH97, LIO98] does not adequately address the flexibility limitations of static hierarchical management frameworks (see Section 2.11), Still, even the latter approach [ZAP99, OLI99] is not suitable for managing networks with dynamically changing topologies and traffic patterns, as they lack mechanisms to change the location where mid-level managers execute. This approach also involves the deployment of a new MA for each introduced monitoring task, which complicates the management of mid-level managers. In addition, critical issues such as criteria for segmenting the network into management domains, explicit determination of the domain boundaries or strategies for assigning mid-level managers to these domains, are not addressed. Furthermore, the architectures described in [LIO98] and [OLI99] are not supplemented by prototype implementations.

In Chapter 6, we introduce a hierarchical MA-based management framework that deploys at runtime mobile mid-level managers (when specific criteria are satisfied), with the ability to dynamically adapt to the managed network dynamics.

3.6.5.2. Mobility Schemes

The usage of MAs covers many aspects in NSM. *Single-hop MAs* are suitable for the easy deployment of software, functionality and services [MAG96]. In particular, single-hop MAs have been used to deploy adaptable instrumentation facilities, compute health functions [PUL00b], provide on-the-fly adaptors for newly installed NEs and services [WHI99]; also to return QoS alarms and periodic summarisation reports by processing raw management data [BOH00c]. In that context, single-hop mobility can be considered as successor of the MbD paradigm.

Multi-hop MAs go one step further, allowing the execution of management tasks on a set of NEs in a sequential manner. Multi-hop mobility may be preferable in several application scenarios (see Section 3.4), for instance when short-term tasks are to be executed over multiple NEs; in such case the code deployment time is reduced, the manager station is not overloaded and the network area around it is not saturated by the simultaneous generation and transmission of agents [BOH00b]. This mobility scheme is, therefore, suitable for collecting on-line data and performing simple monitoring and configuration tasks on several NEs.

Hence, multi-hop MAs have been used to compute health functions [ZAP99, PIN99], return reports related to a set of NEs [SIL99a], discover nodes verifying specific conditions

[PUL00b], perform traffic analysis on remote subnets [FER01], etc. Additional applications include fault diagnosis [ELD99], or collection of data pre-processed by other agents [KNI99, LIO99]. It should be emphasised though that applications using multi-hop agents for collection of data, later processed by the manager station [KU97, SAH98, CHI99, COR98a] do not scale, as not only processing bottlenecks are created at the manager platform, but also the network overhead is increased. Hence, it is essential to benefit from the semantic compression that multi-hop MAs can perform while moving from one host to another, thereby preventing the rapid growth of their state size and minimising the migration overhead [BAL98].

An aspect of management scalability not sufficiently addressed by existing applications relates with the management of large sets of NEs. In such case, launching a *single* multi-hop MA with the task to collect data from all the managed devices can lead to large round-trip delays and also to increased network overhead, even when performing semantic compression of data. In Chapter 5, we address this issue by launching several MA objects, travelling in parallel, with each visiting a relatively small group of devices.

Interestingly, although the potential of using multi-hop agents to perform correlation of data collected along their itinerary has been recognised by researchers [FUG98, LOP00], none of the existing applications implements this principle. MAs can be used to realise a simple data correlation model, providing more even distribution of processing load and returning high-level information to the manager station with no need for further processing. In Chapter 7, we describe an application scenario where MAs can be efficiently used to perform correlation of management data.

Active migration is often combined with a society of MAs imitating ant-like behaviour [APP94, MIN99, SCH97, BON98, WHI98]. Although such kind of MA systems is particularly difficult to manage, they are very suitable for dynamic networks, where random mobility combined with a large number of MAs allows to easily detect changes related to performance, faults or configuration and react accordingly. However, active migration is not necessary for network performance management applications, which represent the main focus of this thesis, as the list of managed devices involved in the monitoring process is typically known in advance. Should new devices are installed, the latter can advertise themselves directly to the manager, without requiring to be discovered by continuously roaming MAs (see Section 4.4.1.3.8). Besides, the use of active migration would imply increased complexity (extra code needed for itinerary control), hence, increased MA size and migration overhead.

Concluding, management applications can benefit either by single-hop or multi-hop MAs (see Section 3.6.3). The selection of the appropriate migration scheme should depend on the type of the application, the managed network size, the duration of the management task's execution, etc. As a result, NSM-oriented MAPs *should support both single-hop and multi-hop agents*. The design of our MA framework, introduced in Chapter 4, satisfies this requirement. Last, but not least, the choice of MA-based solutions instead of alternative approaches management distribution approaches should necessarily be justified through analytical evaluations. The performance of our framework and the applications developed on the top of it, is assessed through extensive quantitative evaluations, validated by experimental results.

3.7. SUMMARY

The advent of MA technology has signalled many potential benefits in the network management arena and attracted the attention of several researchers working on the field. In particular, MAs promise to overcome the limitations of traditional centralised architectures and address the weaknesses of distributed management approaches reviewed in the previous chapter. As a result, MA-based management applications have largely increased in number. The intensity of research activity on that field is strongly related with the proliferation of MAPs expressly developed with management applications orientation. These platforms have demonstrated improved security, fault tolerance, MA control, inter-agent communication and interoperability features, while some also aimed at optimising the performance of MA migrations through sophisticated code distribution mechanisms. All these facts signify that MA community is about to reach the state of maturity.

However, there are a number of issues related to the scalability and flexibility of MA-based management that need to be carefully evaluated. For instance, appropriate organisation models and mobility schemes that meet the specific requirements of management applications need to be identified and evaluated to ensure that MAs are effectively used and offer performance benefits. The investigation of these issues comprises the main part of the research work presented in the following chapters, which describe the design and implementation details of a MAP tailored to management applications, with the main focus being on network monitoring and performance management.

CHAPTER 4

THE MOBILE AGENT FRAMEWORK

4.1. INTRODUCTION

This chapter comprises the introduction to a core Mobile Agent (MA) framework design and implementation issues. The framework is intended for use in Network & Systems Management (NSM) applications, hence, a number of management-related issues have been taken into account and several optimisations have been performed to achieve NSM orientation. The presented infrastructure exploits the benefits of MAs to carry out semantically rich management operations in a highly scalable, flexible and efficient manner.

The design and implementation ideas introduced in this chapter have been originally published in the proceedings of the *IEEE International Conference on Communications* (ICC'99). The description of several extensions have been included in papers published in the proceedings of the *4th IEEE International Symposium on Computers and Communications* (ISCC'99) and the *3rd International Workshop on Intelligent Agents for Telecommunication Applications* (IATA'99). Full references are given in Appendix A.

The remainder of this chapter is organised as follows: Section 4.2 highlights the main goals identified for the framework's design, with special focus on the particular needs and requirements of management applications. Section 4.3 summarises our Mobile Agent Platform (MAP) design. Section 4.4 comprises the core of this chapter, describing the implementation of the introduced infrastructure. The features of the main building components are presented in detail. Special focus is given on the framework's fault tolerance features and its class loading mechanism. A quantitative evaluation of the framework is presented in Section 4.5, where simple mathematical formulations are devised to model the response time and network overhead of centralised and MA-based management.

To thoroughly evaluate the performance of the proposed framework, a number of timing and network overhead experiments have been conducted and presented in Section 4.6. A first set of measurements compares the performance of MAs against that of Java Remote Method Invocation (RMI) both in terms of response time and network overhead. Following that, we present experiments that evaluate the dependency of multi-hop MAs response time on the amount of collected data and the itinerary length. The empirical results of these experiments are contrasted with the theoretical findings of the quantitative evaluation and ways for improving the MA framework's performance are identified. Finally, Section 4.7 summarises and draws the conclusions of this chapter.

4.2. Main Goals of the Mobile Agent Framework Design

The following points have been identified as the main goals regarding the design and development of our MAP:

- *Network management applications orientation*: Unlike publicly available general-purpose MAPs, the introduced architecture should take into account the special characteristics and requirements of NSM applications and therefore provide a flexible, scalable and lightweight environment tailored to management operations. In addition, a variety of features typically offered by general-purpose MAPs may be omitted as they do not add any value to an NSM-oriented framework, while increasing the complexity of the platform and its requirements on system resources.
- *Mobility support*: The ability of objects to move from host to host offers a powerful abstraction that should not be disregarded when building distributed management applications. The architecture should therefore provide a set of services to allow the *migration* of management agents, regardless of the underlying management models. *Cloning* of MA objects should also be supported in order to allow instant creation of MAs, when it is required to share the responsibility of a management task among multiple MAs with identical functionality or minimise MAs deployment latency.
- *Modularity*: Maintaining an architecture with intrinsic modularity eases the addition of new services or the modification of existing ones.
- *Support for existing management standards*: Our architecture encompasses the dominant NSM framework of Internet world, i.e. the Simple Network Management Protocol (SNMP). Due to its huge installation base, integration with SNMP was considered of vital importance to maintain compliance with legacy management systems. The importance of

integrating MA-based management environments with SNMP is also discussed in [SIM99, PAG00].

- *Support for 'disconnected' operations*: Management operations dependency on network resources should be minimal, in other words, it is essential that management tasks execution is not vulnerable to link or node failures or sensitive to traffic conditions. The required support for disconnected operations can be guaranteed by using autonomous MA entities, which are able of performing their decentralised management tasks without requiring communication with a central manager station. These MAs should be able to continue their execution even when the communication link with the manager is disrupted or the manager station itself fails.

- *High performance*: A variety of network performance management tasks have very demanding time requirements. It is therefore of critical importance to design an infrastructure that guarantees prompt execution of time sensitive monitoring tasks, allowing sophisticated Network Management Systems (NMS) to foresee possible congestions or failures and take preventive measures before the actual error occurs. As a result, MA migrations should be optimised in terms of their latency and MA objects assigned the highest priority among the processes executing on the same system. In addition, the itineraries of multi-hop MAs carrying out management tasks should be optimally designed so as to minimise the overall response time.

- *Scalability*: Scalability is of major concern when designing distributed management applications. MA-based approaches in NSM promise scalable solutions, however frequent transfers of MAs may cause excessive use of network bandwidth and result in poorer performance than centralised frameworks. Hence, it is essential to ensure that MA migrations take place only when they are absolutely necessary, whilst imposing minimal burden on network resources when they occur.

- *Lightweight footprint*: The applications required to receive, instantiate and launch MAs (Mobile Agent Servers) should be designed so as to be as lightweight as possible and therefore provide execution environments that can be installed on any network device, including those with limited storage capacity and processing capabilities.

- *Security*: One of the main arguments commonly used against MAs is the security threats this technology brings forth. Hence, security concerns should be carefully taken into account, with the designed architecture offering protection against MA attacks and the MA objects shielded from tampering and eavesdropping. In addition, access control on MAs actions on system resources and services should be supported.

- *Fault tolerance*: A MAP should be able to cope with situations where link or node failures disrupt the normal migration process of roaming MA objects. Fault tolerance features should also cope with the scenario where the failed node is the manager station itself and ensure that the valuable management information collected by the MAs is not lost.
- *Dynamic class loading*: Existing management tasks often need to be modified. These actions should be carried out in a user-transparent manner, without disrupting the NMS operation. That could be achieved through a network class loader (CL) installed on every managed device, able to receive at runtime the definition of an MA object representing a management task. CLs should allow the NMS not only to introduce new services, but also to replace existing ones; in other words, CLs should be able to distinguish different versions of the same MA class.
- *Ease in introducing new services*: It is essential to provide an open architecture in which the administrator can easily add new services at runtime. In our framework, there is a direct mapping of management functions to certain MA objects. The effortless introduction of new services is accomplished through a graphical tool that automates MA code generation and eases the deployment of new MAs that carry out specialised management operations.
- *Platform Independence*: Last, but not least, the implementation of a framework that supports any network device, regardless of the underlying hardware platform or operating system, is crucial in order to cope with the heterogeneity of modern multi-vendor networking environments. The “write once - use everywhere” slogan of Java promises to achieve the required degree of platform independence.

4.3. Overview of the Introduced Mobile Agent Platform

4.3.1. Implementation Language

The introduced platform is entirely developed in Java due to its inherent platform independence that makes it portable across distributed heterogeneous environments. This and other features, highlighted in Section 2.7.1.1 and 3.5.2, suggest Java as a suitable platform for developing MA-based applications. Hence, the selection of Java as implementation language for our framework has been a natural choice. In fact, the vast majority of MAPs developed over the past few years are implemented in Java (see Sections 3.5.6 and 3.6.1.).

The framework’s source code has been developed using the Java Development Kit (JDK) 1.1 library, which was the most recent JDK version at the time the implementation of the framework commenced. Although the compatibility of the prototype with the JDK’s latest version (1.2) has not been tested, it is expected that only minor changes are required to achieve full compatibility.

4.3.2. Framework’s Overview

The framework introduced in this chapter provides the basic functions required to program MA-based applications. Nevertheless, several optimisations have been incorporated during its implementation to achieve a flexible design that not only has minimal requirements on distributed system and network resources, but also integrates special features particularly suited for NSM applications. The reasons that led to the decision to implement a new MAP from scratch rather than using one of the many available general-purpose MAPs, have been highlighted in Section 3.5.6.

In our platform, the interface between the visiting agents and legacy management systems is achieved through the Mobile Agent Servers (MAS). A MAS provides incoming MAs entry points to services, represented by Java objects within the MAS. Security issues have also been addressed including authentication of visiting MA objects, encryption of sensitive management information and authorisation of MA actions.

A manager application has been developed to co-ordinate the policies of monitoring and controlling the Network Elements (NE). In this chapter, we investigate the performance of the “standard” use of MAs, i.e. multi-hop MAs, that is the agent code migrates *sequentially* between managed devices with data processing performed locally. This alleviates the need for broadcast polling and can result in a significant reduction in NSM data at the source, as only high-level management information is delivered to the manager.

An MA is an instance of a user-defined Java class. This class must inherit the properties of a generic class that provides all the core functionality to implement mobility. The platform implements *weak mobility*, namely the MA can carry its persistent state but not its execution stack. The implementation of *strong mobility*, apart from representing a non-trivial task as explained in Section 3.5.5.1, is not considered essential for performance management applications. That is, MAs are typically required to start execution from a specified entry point (method) rather than resuming their operation from the point their execution was interrupted on the previously visited host [CAB00].

Unlike all known publicly available MAPs, which involve the transfer of both the agent’s code and state information on every migration, the framework introduced herein adopts a more lightweight mechanism whereby the bytecode is transferred to each managed device only once, at the MA construction time (we assume that NEs have fairly large storage capabilbity); following that, the transfer of the MA state is sufficient to carry out distributed management tasks. That results in moderating the demand on network resources, since bytecode size is typically much larger than state size [BAL98].

We also introduce a novel tool prototype, the Mobile Agent Generator (MAG), which creates MAs in response to service requirements. Such MAs may be generated post MASs initialisation to accomplish intelligent management tasks, tailored to the needs of a changing network environment. Thus, the MAG realises a flexible infrastructure through the creation of new MA instances, which dynamically extend NMS functionality.

The introduced prototype is logically structured in *7 packages* [ARN96], each of which includes a number of classes implementing a specific part of the framework. A short description of these packages is given in Table 4.1.

Package	Number of classes	Description
`GUIs`	21	Implements the framework's user interfaces
`Manager`	15	Implements the manager application and its building sub-components
`MAS`	12	Implements the Mobile Agent Server and its building sub-components
`MCode`	10	The MA 'base' class, its service-oriented sub-classes and several classes for manipulating MA objects
`RMI`	6	Implementations of the RMI servers and their respective interfaces
`Security`	2	Implementation of cryptography algorithms and a customised SecurityManager
`Utilities`	6	Various utility classes

Table 4.1. Structuring of the prototype in packages

Full details on the MAP's building blocks implementation are given on Section 4.4.

4.4. Infrastructure Overview - Implementation Details

4.4.1. The infrastructure's main building blocks

The infrastructure consists of the following major components (see Figure 4.1):

- The Manager, responsible for launching and controlling MAs and displaying results;
- The MA objects, capable of migrating between the managed entities to collect information based on pre-defined policies;
- The Mobile Agent Server (MAS), capable of receiving MAs and providing an interface to the local physical resources;

- The MAG, a factory that generates service-oriented MA objects.

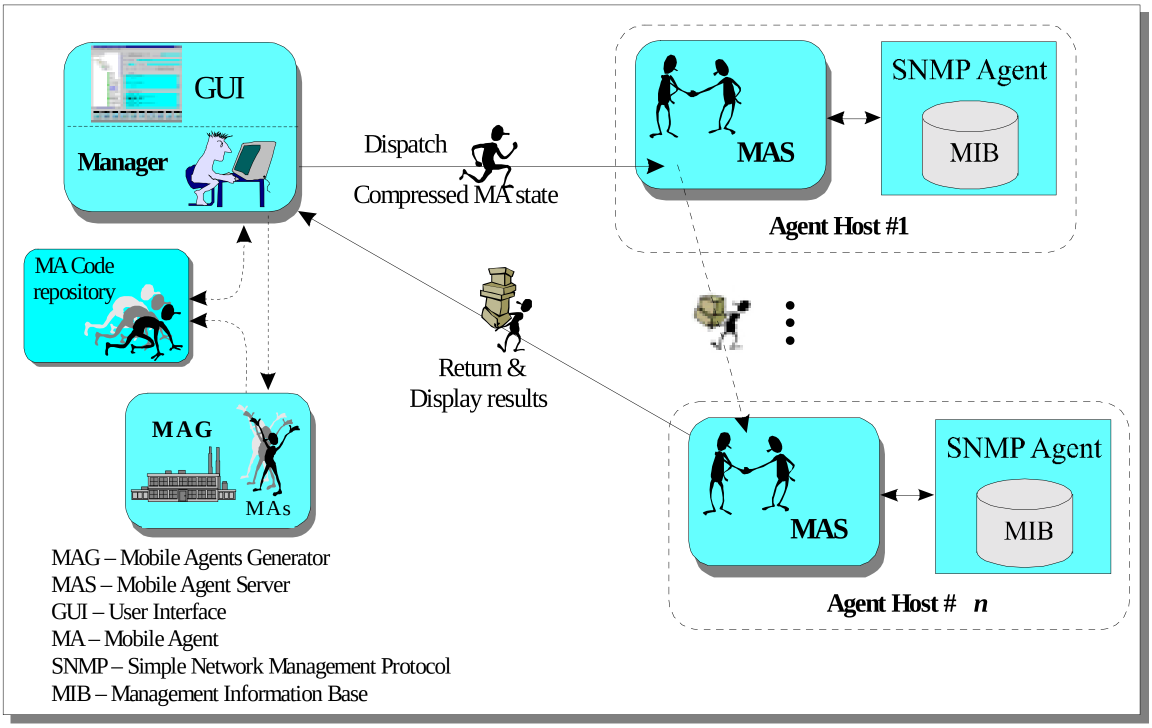


Figure 4.1. The Mobile Agents-based Infrastructure

The following sections elaborate on the design and implementation details behind its one of these components.

4.4.1.1. Manager application

The manager application performs monitoring and control operations through interacting with devices running agent processes. It composes a multithreading environment where a main thread instantiates, controls and co-ordinates the operation of a number of specialised threads (see Figure 4.2).

The manager maintains a 'discovered' list of active MASs. In particular, at startup time the *Network Discovery* thread parses a text file ("*network configuration*" file), which includes a list of the managed devices' IP addresses and information regarding the subnets they are physically connected to; it then 'pings' the corresponding hosts checking whether there is an active MAS server thereon. It is noted that active servers are discovered either at manager initialisation or whenever a new MAS starts operation. In the latter case, the manager application is notified and the host name appended to that manager's 'discovered' list. Clearly, the discovery of active agent servers based on parsing a text file is not sufficiently flexible and dynamic (it does not adequately deal with topology changes), however the design and implementation of a topology discovery algorithm was beyond the scope of this research work. The implementation of an automated naming service is not a trivial task, as it requires configuration information obtained from various sources. That includes the Management Information Bases (MIB) of routers, bridges and switches, use of Internet Control Message Protocol (ICMP) messages, the Domain Name Service (DNS), etc. [LIN99].

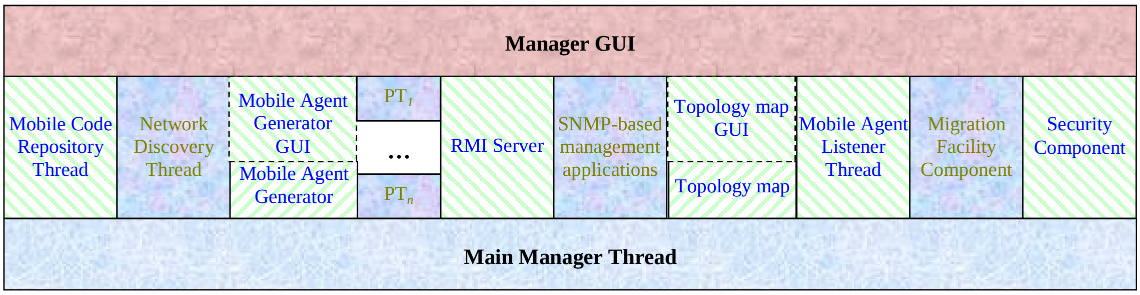


Figure 4.2. Break-down of the components that compose the manager application

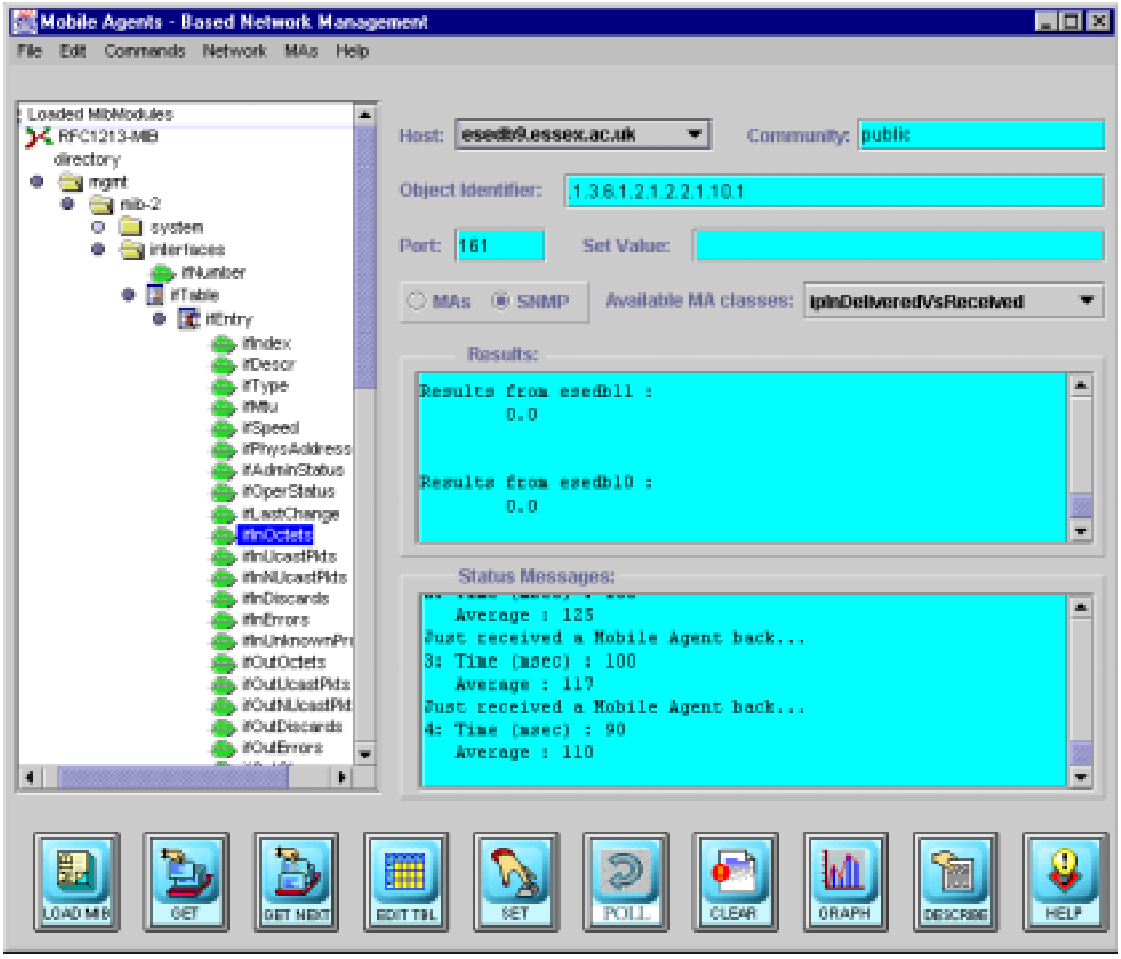
In the event of a pending MA-based task, the manager retrieves the MA's definition from a Mobile Code Repository (MCR), it instantiates an MA object and assigns it an itinerary including all active agents hosts, unless a travel plan is manually specified. These actions are basically performed by one of the *Polling Threads*, whose operation is described in Section 5.2.1.1. The MA's state information is then compressed (using the Java *gzip* utility) and transferred through the Migration Facility Component (MFC) to the first destination host. When the MA returns back, it is received by the Mobile Agent Listener (MAL) thread, which retrieves the results carried by the MA and presents them to the user. The MAL and MFC components are basically identical to the ones used by the MAS entities; their implementation details are discussed in Sections 4.4.1.3.1 and 4.4.1.3.5, respectively.

The manager application is equipped with a user-friendly Graphical User Interface (GUI) including a MIB browser, 'event' and 'results' panels, a pull-down menu, several action buttons, etc (see Figure 4.3). The Manager's GUI also displays time statistics regarding the response time of completed management tasks; these statistics can optionally be written in a text file (this action is controlled through the GUI's menu), in a format easily extracted and inserted in a spreadsheet for further processing and analysis. Among the possible future extensions of the framework, is its integration with the JDBC[1] driver to allow automatic logging of operational results in a database that could be later used to generate high-level reports or graphs. Another direction of future work will include the design and implementation of an API used for developing manager applications, which will offer the manager GUI as an optional feature.

[1] The Java Database Connectivity (JDBC) [JDBC] kit allows Java programs to connect to any commercial relational database (given that it provides JDBC drivers) and query or update it using the industry standard query language (SQL). In the NSM context, JDBC makes it easy for Java-enabled management platforms to store or retrieve management data from databases and possibly present it to the administrator in a user-friendly manner.

Currently, additional facilities allow:

- polling of agents for specific object values; the fluctuation of arithmetic values may optionally be graphically illustrated using Java-enabled graphs updated in real-time (see Figure 4.4);
- initialisation of automated network discovery processes;
- acquisition of on-line MIB variable descriptions (see Figure 4.5);
- initialisation of MA-based or SNMP management operations. MA-based operations will be described in detail in the following sections. SNMP operations have been supported in order to compare their performance against MA-based solutions. These operations include simple 'get', 'get-next' and 'set' requests, retrieval of SNMP tables and periodic, synchronous/asynchronous polling of an arbitrary number of NEs for a set of MIB objects.



Host: esedb9.essex.ac.uk

Community: public

Object Identifier: 1.3.6.1.2.1.2.2.1.10.1

Port: 161

Set Value:

○ MAs ● SNMP

Available MA classes: ipInDeliveredVsReceived

```
Results from esedb11 :
        0.0


Results from esedb10 :
        0.0
```

```
   Average : 125
Just received a Mobile Agent back...
3: Time (msec) : 100
   Average : 117
Just received a Mobile Agent back...
4: Time (msec) : 90
   Average : 110
```

Figure 4.3. The manager application Graphical User Interface

Several classes of the Java-based AdvnetNet SNMP package [AdventNet] have been utilised to built high-level SNMP applications and also as a basis of the real-time line graph and MIB browser components.

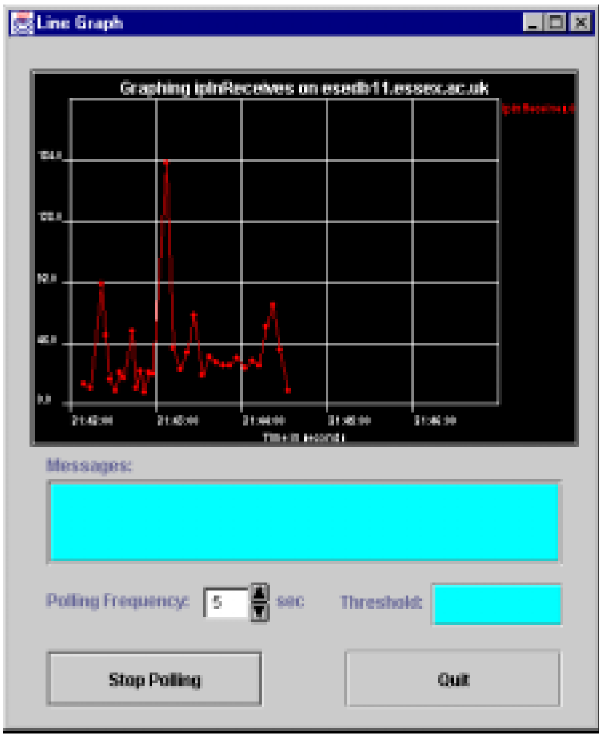
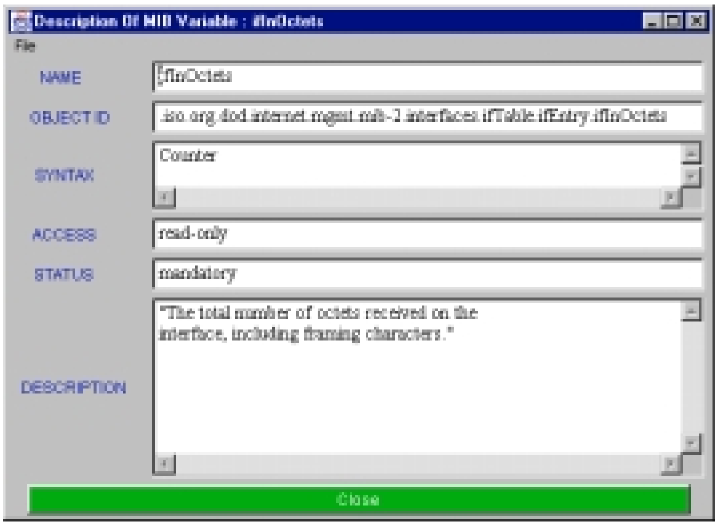


Figure 4.4. Polling of a MIB variable



NAME: ifInOctets

OBJECT ID: .iso.org.dod.internet.mgmt.mib-2.interfaces.ifTable.ifEntry.ifInOctets

SYNTAX: Counter

ACCESS: read-only

STATUS: mandatory

DESCRIPTION: "The total number of octets received on the interface, including framing characters."

Figure 4.5. On-line acquisition of a MIB object description

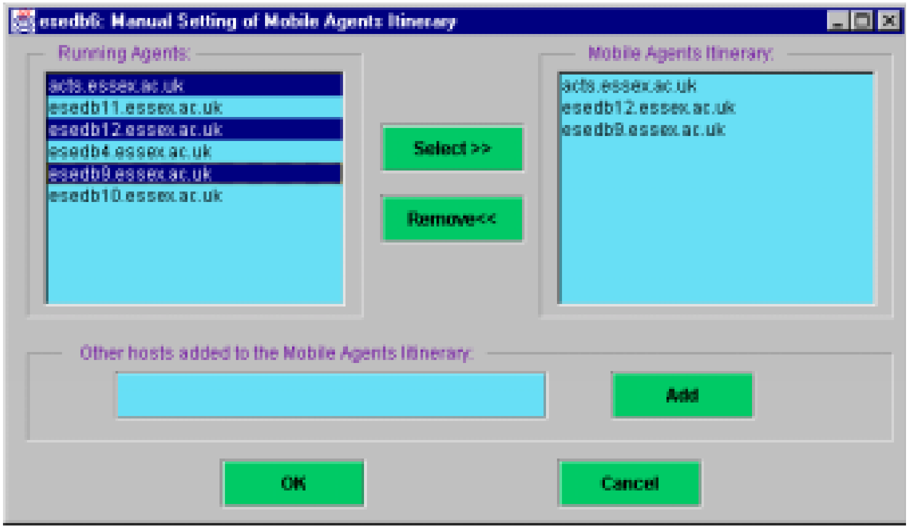
The Manager application also starts and controls an RMI server; the role of this component will be discussed in Sections 5.2.2.1 and 6.3.2.

4.4.1.2. Mobile Agent Implementation

An MA object is identified by its code (description of its behaviour), state information (modifiable variables) and attributes (static/permanent information). In the context of management-related applications, MAs are Java classes supplied with:

- a unique ID string, which serves for distinguishing an MA object from others (the development of a naming architecture was beyond the scope of this work, therefore, the ID string is simple structured in a <origin_host:serial_number> format);
- an 'itinerary folder', including a list of hosts to be visited during the MA's operational travel; the itinerary may either automatically built or manually specified by the user (see Figure 4.6).
- a 'data folder', used to store collected data;
- a 'problems folder', used to report faults;
- a flag denoting the transport protocol used for the MA's transfers;
- a flag indicating whether encapsulated data should be encrypted or not;
- a byte array containing the MA's 'signature', used for authenticating the MA when it arrives at a destination host;

Itinerary, data and problem folders have been implemented as Java *vectors*, i.e. dynamic arrays (`java.util.Vector` class). At each visited NE, a data sample is typically appended to the data folder whereas the local NE address is removed from the itinerary vector. In this way (i.e. by reducing the itinerary vector size) and by applying data aggregation methods, we prevent the MAs state from growing too rapidly. MAs state is compressed (using the Java *gzip* utility) before their transfer to the next host to minimise communication overhead. In addition, the transfer of MAs over both TCP and UDP transport protocols has been implemented.



esedb6: Manual Setting of Mobile Agents Itinerary

Running Agents:
acts.essex.ac.uk
esedb11.essex.ac.uk
esedb12.essex.ac.uk
esedb4.essex.ac.uk
esedb9.essex.ac.uk
esedb10.essex.ac.uk

Mobile Agents Itinerary:
acts.essex.ac.uk
esedb12.essex.ac.uk
esedb9.essex.ac.uk

Other hosts added to the Mobile Agents Itinerary:

Figure 4.6. Manual setting of a Mobile Agent's itinerary

MA classes also include a number of methods that facilitate the interaction with polled devices. The MA code is minimised in order to reduce bandwidth requirements. Through the process of *serialisation*, the state of an MA object can be saved, transferred through the network and reconstructed (de-serialised) at the receiving node (see Section 3.5.2).

In order to improve design flexibility, maximise code reuse and accelerate prototype development, an MA 'superclass' has been implemented, providing the root attributes listed above, which are essential for any MA object. The `MCode.MA` superclass implements the `java.io.Serializable` interface[2], allowing instances of the class to be serialised/de-serialised. Any class intended to employ mobility characteristics should necessarily sub-class the MA superclass. The methods of the MA superclass are shown in Figure 4.7.

```
public final void setParams(String ID, String creator, Vector itinerary, boolean TCP, String type,
                    boolean encryption, byte[] sign){}
/* This method sets the MA's parameters; it can be invoked ONLY by the MA's creator (manager),
Otherwise, an Not_Authorised_To_Initialise_Exception exception is thrown. */

public void run(){}   // The main execution block for this MA
public void provideHandle (MAServer MaServer){}  // Binds the MA object to the local MAS entity
public final String getID(){} // Returns the MA's ID
public final String getCreator(){}  // Returns the name of the host where the MA was originally created
public final boolean getTransmissionProtocol(){}   /* Indicates the transport protocol used for the MA
                                                            transfers (true for TCP, false for UDP) */
public final Vector getData(){}  // Returns the data collected by the MA
public final byte[] getSignature(){}  // Returns the "signature" of the MA (used for authentication)
protected final void encapsulateData (Object value){}    /* Inserts a new sample of data into the data
                                                                 vector */
public final Vector getProblemsFolder(){}  /* Returns the names of the hosts that could not be visited
                                                      during the MA's travel */
public final InetAddress getNextHost(){} // Returns the next host included into the MA's itinerary
public void onStart(){}   // Invoked when the MA is first instantiated
public void onArriving(){}  // Invoked when the MA arrives at a host
public void onMoving(){}    // Invoked when the MA is about to be launched
public void onFailedMoving(String host) {}  /*    Invoked when the MA is not able to migrate to a
                                                         specific host */
public void onStop(){}   // Invoked when the MA thread's execution is about to stop
public void onSuspend(){}   // Invoked when the MA thread's execution is about to be suspended
public void onResume(){}  // Invoked when the MA thread's execution is about to be resumed
```

Figure 4.7. The methods of the MA super-class

The `setParams()` method is invoked by the manager application to set the MA's basic parameters (e.g. ID, itinerary, etc). To protect MAs against tampering, sensitive MA properties may be specified only *once*, when the MA is created. If a malicious host attempts to modify these properties, a `Not_Authorised_To_Initialise_Exception` exception is thrown. There are also methods used by the MA itself, for instance to obtain the name of the next host to be visited or request migration. Another set of methods is used to facilitate the interaction with visited hosts, for example to bind the MA to the local MAS server, encapsulate a sample of data into its data folder, return the MA's ID, signature, data, etc. A last set of

[2] A Java class cannot be serialised unless it implements the `java.io.Serializable` interface, or it extends another class implementing the `java.io.Serializable` interface.

methods are automatically invoked when the MA object is instantiated, arrives or moves from a host, fails to migrate, or when its execution is started, stopped, resumed or suspended.

In addition, we have implemented several service-oriented classes (discussed in Chapter 7) that extend the MA superclass. These in turn, are sub-classed by MA classes created by the MAG tool. This flexible hierarchical approach minimises MA bytecode and eases the creation of service-specialised MAs.

4.4.1.3. The Mobile Agent Server: Interface to Managed Resources

The interface between visiting MAs and legacy systems is implemented through MAS modules (Figure 4.8); we assume that a MAS server is installed on every managed device. Functionally, the MASs reside above standard SNMP agents, defining an efficient run-time environment for receiving, instantiating, executing, and dispatching incoming MA objects, whilst protecting the system against malicious agents attacks. The SNMP agent process is started automatically at MAS initialisation, if not already active. The port in which the SNMP agent listens for incoming requests can be specified by the user in the command line, otherwise the default UDP port 161 is used. In Windows platforms, the standard *SNMP service* provided with the operating system is used. In UNIX stations, the *snmpd* daemon that comes with the *UCD-SNMP* package [UCD-SNMP] has been utilised. This package is originally based on the earlier CMU-SNMP implementation, but has been greatly enhanced, ported and fixed so that it supports, among others, Solaris, Linux and HP-UX platforms.

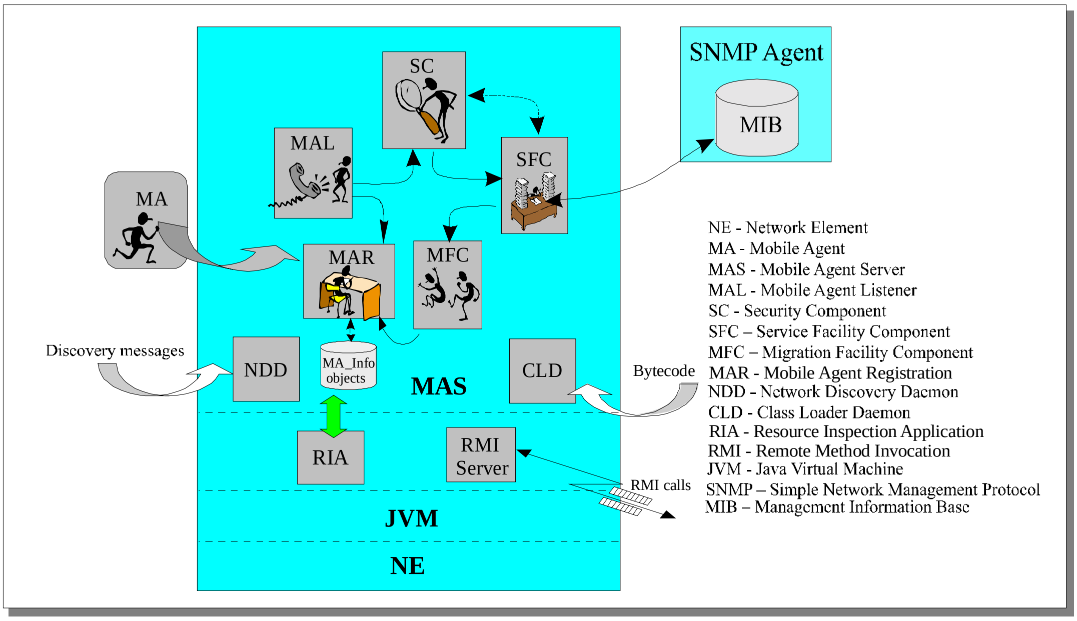

SC
MAL
SFC
SNMP Agent
MIB
MA
MAR
MFC
Discovery messages
NDD
MA_Info objects
MAS
CLD
Bytecode
RIA
RMI Server
RMI calls
JVM
NE
NE - Network Element
MA - Mobile Agent
MAS - Mobile Agent Server
MAL - Mobile Agent Listener
SC - Security Component
SFC – Service Facility Component
MFC – Migration Facility Component
MAR - Mobile Agent Registration
NDD - Network Discovery Daemon
CLD - Class Loader Daemon
RIA - Resource Inspection Application
RMI - Remote Method Invocation
JVM - Java Virtual Machine
SNMP – Simple Network Management Protocol
MIB – Management Information Base

Figure 4.8. The Mobile Agent Server

Similarly to the manager application, the MAS class is designed so as to compose a multi-threaded environment. Apart from increasing the framework’s modularity, this scheme allows independent components to work concurrently and therefore decrease the overall delay when handling MA objects, especially when the rate of incoming MAs is relatively high or bursty. For instance, a MAS may receive or authenticate an MA at the same time that a second MA is dispatched to another host. The main building blocks of a MAS server are illustrated in Figure 4.8 with their implementation detailed in the following sections.

4.4.1.3.1. Mobile Agent Listener (MAL)

The MAL is a daemon, which basically includes two separate threads: the first (`MAS.TcpListener` class) listens for incoming MAs on a well-known TCP port, with the second (`MAS.UdpListener` class) listening on an advertised UDP port. To illustrate, let us focus on TcpListener's operation. TcpListener opens a TCP socket and blocks itself on listening mode waiting for incoming MAs. Upon the arrival of an MA, its state is decompressed and de-serialised. The MA is then authenticated by the Security Component (SC). If the authentication is successful (the MA has been created by a trusted host), the MA is first registered by the Mobile Agent Register (MAR) component. A separate thread is subsequently created for the MA's execution and assigned the *maximum possible priority* (`Thread.MAX_PRIORITY`). That way, MA threads are always prioritised against other pending executing threads on the local device, ensuring timely execution of their monitoring tasks.

TcpListener then binds the MA object to the local MAS server (it provides it a reference of the MAS class instance) to enable the interaction of the two parties (allow the MA to invoke MAS methods and vice-versa). Last, the MA's execution is started (its `run()` method is called). From that point onwards, the execution of TcpListener and MA threads is detached, with the first returning back to listening mode. The process of decompression and de-serialisation is carried out by the `receive()` method of the `TcpListener` class. The latter is called by its `run()` method which also controls the MA's authentication and registration process and starts its execution. The implementation of these two methods is presented in Figure 4.9. UdpListener's operation only differs on that compressed and serialised data are retrieved from a byte array, which forms the payload of a received UDP packet.

```
 // This is the main execution block
public void run() {
    while (true) {
        MA ma = receive();   // Receive (de-serialise) the MA object

        if (ma != null){
            // Authenticate the received MA
            if (sc.Authentication(ma)) {
                // Create a new thread for the MA's execution
                Thread MaThread = new Thread(mas.MAsThreadGroup, ma, "MA");
                // Asign the MA object the maximum priority
                MaThread.setPriority(Thread.MAX_PRIORITY);
                ma.provideHandle(mas); // Bind the received MA to the local MAS server
                MaThread.start();  // Start the MA's thread execution
                // Ask the Mobile Agent Register to register the received MA
                mar.keepMaReport(ma, MaThread);
            }
        }
    }
}

// --------------------------------------------------------------------

// Receives and de-serialises an MA object
MA receive() {
    // Establish a TCP socket that can queue up to 3 MA objects
    ServerSocket serSock = new ServerSocket(MA_TcpPort, 3);
    Socket connection = null;
    MasObjectInputStream = null;

    // Block to listening mode waiting for incoming MA objects
    try {
        connection = serSock.accept();
        connection.setTcpNoDelay(true);  // Do not use Nagel's algorithm

         // Obtain a reference to the socket's data input stream
        InputStream is = connection.getInputStream();
           /*  Redirect the input stream into a zip stream and then into an MAs' de-
                                     serialisation stream. */
        dataIn = new MAsObjectInputStream
                              (new GZIPInputStream(new BufferedInputStream(is)));

         // De-compress and se-serialise the MA's state
        try {
            ma = (MA)dataIn.readObject();
        } catch (ClassNotFoundException exc) {
             System.err.println("Problem in deserialisation: " + exc.toString());
        }
        connection.close();
        return ma;
    } catch (IOException e) {
        try {serSock.close(); connection.close();}
        catch (Exception e1) {
            System.err.println("Socket has been already closed");
        }
      System.err.println("Exception while listening for MA connections : " + e);
    }
}
```

Figure 4.9. TcpListener's `run()` and `receive()` methods

4.4.1.3.2. Security Component (SC)

This component acts as the system's protective barrier. Specifically, the SC verifies the authenticity of the received MA through the use of security keys[3], ensuring that only trusted agents, dispatched by authorised hosts, are instantiated. In addition, it authorises the actions performed by the locally executing MAs and insures the privacy of sensitive management data returned to the manager station. The RSA (Rivest-Shamir-Adleman) algorithm [RIV78], based

[3] *Keys* represent secret information used by cryptographic algorithms. In traditional cryptography, the sender and receiver of a message know and use the same secret key (symmetric cryptography); the sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. *Public-private* key algorithms use a different key for encryption and decryption. The generation, transmission and storage of keys are termed *key management*.

on the ‘*public-private pair of keys*’ paradigm, has been implemented providing both *authentication*[4] and *encryption*[5] features. The reasons for choosing RSA rather than other authentication (e.g. SHA-1, DSA, MD5) or encryption (e.g. DES) algorithms was that the design and implementations of RSA authentication and encryption algorithms were very similar; that characteristic helped to maximise code re-use and enabled rapid prototype development. RSA’s security strength is related to the assumption that detection of the private key (needed to sign or decrypt the transmitted data) is highly improbable. In particular, the security of RSA relies on the difficulty of factoring large integers (dramatic advances in factoring large integers would make RSA vulnerable). In the current prototype, the encryption and authentication algorithm implementations are included in the `Security.Security` class, whose method definitions are shown in Figure 4.10.

```
// Encrypts the original message
public static byte[] Encryption(String message);
// Decrypts the encrypted message
public static byte[] Decryption(byte[] encrMsg);
// Generates a digital signature of the original message
public static byte[] Signature(String message);
// Checks whether the original message is recovered after processing its sugnature
public static boolean Authentication(byte[] signature, byte[] original);
```

Figure 4.10. The method definitions of the `Security` class

Thus, the steps followed during an MA object round-trip are:

- The manager application signs the MA using the public key. The digital signature generated is stored in a byte array, encapsulated and sent along with the MA.
- Upon arrival, the SC executes the authentication function in conjunction with the private key and verifies that the MA state has not be tampered (this ensures data integrity). If the authentication is successful the MA’s execution is initiated, otherwise the manager is informed accordingly.
- The data encapsulated into the MA’s state are first encrypted using the public key, given that data encryption has been requested by the administrator at the MA’s creation time.
- The manager decrypts the data received by invoking the decryption method, which makes use of the private key.

[4] *Authentication* is defined as the action of verifying information in terms of identity or ownership.

[5] *Encryption* is the transformation of data into a form that is as close to impossible to read without the appropriate knowledge (a key). Its purpose is to ensure privacy by keeping information hidden from anyone for whom it is not intended, even those who have access to the encrypted data. *Decryption* is the reverse of encryption, i.e. the transformation of encrypted data back into an intelligible form.

The integration of the SC with the Java Cryptography Architecture (JCA) [JCA], which refers to a generic framework for accessing and developing cryptographic functionality for the Java Platform, will be considered in future extensions of the framework. JCA encompasses the parts of the JDK 1.2 Security API related to cryptography, as well as a set of conventions and specifications. However, only authentication algorithm implementations are provided in the current release. JCA also introduces a *provider* architecture [Providers] that allows for multiple and interoperable cryptography implementations. Implementation interoperability means that various implementations can work with each other, use each other's keys, or verify each other's signatures.

Authorisation: The security of MAS entities has been strengthened by introducing *authorisation* features that restrict the authority domain of visiting MAs on legacy systems. Specifically, the Java standard *Security Manager* (SM), (`java.lang.SecurityManager` class), has been extended to prevent MAs from directly reading/writing files, creating CLs or sub-processes, shutting down the MAS application, etc. The identification of illegal actions is achieved through registering the incoming MA threads to a given *thread group*, i.e. a batch of threads that eases the manipulation of active MAs. Based on the fact that an MA cannot change the thread group it belongs to, whenever a malicious action is detected, the SM checks the thread group of the action's originating thread; if this thread belongs to the MAs thread group, the action is not permitted.

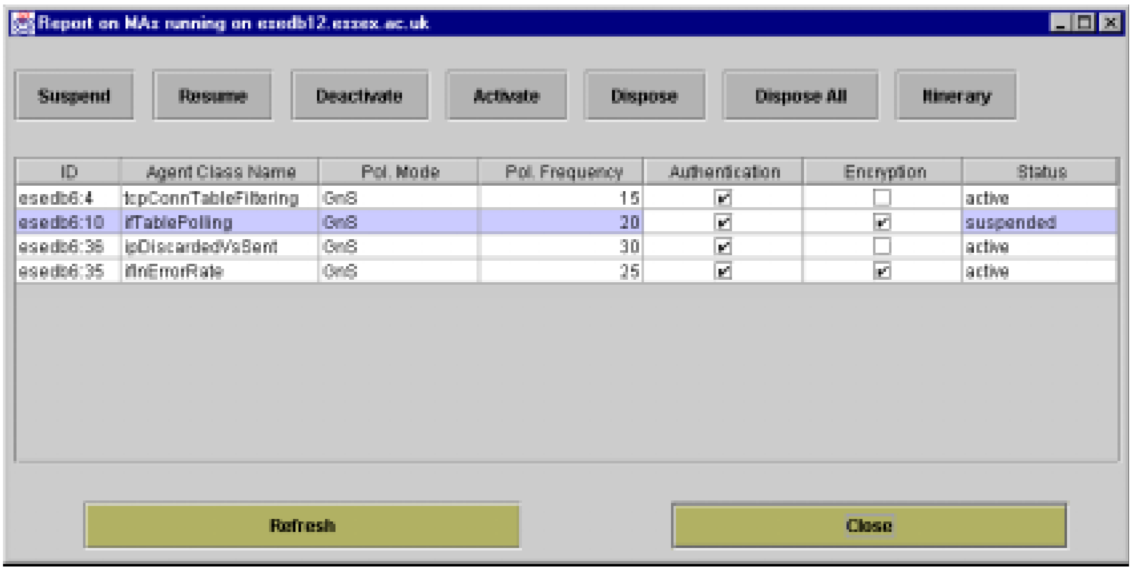
An issue not addressed by the current implementation of the SM is that MAs cannot be restricted from entering an endless loop (consuming all CPU cycles), or creating thousands of new threads. However, in such an emergency situation, the SM could selectively destroy all threads not belonging to the MAS components' thread group. Alternatively, a resource accounting interface like *Jres*[6] [CZA98], a scheme for controlling allocation of different kinds of resources (CPU, memory, etc) within the Java runtime system, could be used.

In general, the implementation of an impenetrable security shield, has not been a principal aim during the development of our framework. Besides, agent security is a particularly complex subject and represents an active and evolving research area [VIG98]. However, with the security precautions described in this section, we consider the network devices to be reasonably safe from malicious MA attacks.

[6] Jres allows per-thread accounting of heap memory, CPU time usage and the number of bytes sent and received through a network connection. In addition to tracking resource consumption, tasks can be informed when new threads are created. Another interesting feature of Jres interface is that it provides mechanisms for setting limits to resources available to individual threads and associates an overuse callback method that is called whenever this limit is about to be exceeded [CZA98].

4.4.1.3.3. Mobile Agent Register (MAR)

MAS entities keep active control of the MA objects executing on their local devices. In particular, the MAR component maintains a hashtable[7] (`java.util.Hashtable`), using MA IDs as a primary key and including a list of "*MA_Info*" objects, each mapped to an MA object running on the local host. MA_Info objects are basically records that include a reference to the MA object they correspond to (allowing the MAS server to perform a number of actions upon it, i.e. invoke its methods) and additional information related to the MA, such as the MA's class name, the execution frequency of its management task, its arrival time, its execution status (activated, de-activated, suspended), etc. The hashtable is dynamically updated whenever an MA arrives or leaves the host. Specifically, the MAL thread inserts a new record when receiving an incoming MA, while the MFC thread deletes the corresponding record as soon as the MA object migrates to another host.



ID	Agent Class Name	Pol. Mode	Pol. Frequency	Authentication	Encryption	Status
esedb6:4	tcpConnTableFiltering	GnS	15	☑	□	active
esedb6:10	ifTablePolling	GnS	20	☑	☑	suspended
esedb6:36	ipDiscardedVsSent	GnS	30	☑	□	active
esedb6:35	ifInErrorRate	GnS	25	☑	☑	active

Figure 4.11. On-line visual profiling of Mobile Agent objects executing on a network device

The information stored within the hashtable can be remotely retrieved through an RMI call (the `getMAsReport()` method of the MAS's RMI server is invoked and a vector of MA_Info objects is returned). A report about the MAs executing on any remote device is then presented to the network administrator through the GUI shown in Figure 4.11. The administrator may then select any of the MA instances presented in this GUI and perform a number of actions upon them. In particular, he/she may suspend, resume, activate an MA or modify its itinerary; these actions are carried out through RMI calls (see Figure 4.14) that cause the automatic update of the remote hashtable and the invocation of a corresponding MA's method.

4.4.1.3.4. Service Facilitator (SF)

Upon successful authentication, the MA is activated and provided a handle to the Service Facilitator (SF) component, which serves as an interface between MAs and services offered to them. SF generally includes the ‘know-how’ of the services offered to incoming MAs, i.e. all the functionality needed by the MA objects to perform their decentralised management tasks. An alternative approach would be to integrate this functionality within the MA classes themselves; however, this would result in the unnecessary transfers of large pieces of code that would seriously overload network resources.

At its current implementation, the SF component is solely oriented to NSM applications; as such, it basically offers a library of methods that facilitate the interaction of MA objects with the local SNMP agent. In particular, it allows the MAs to perform from simple SNMP requests (‘get’, ‘get-next’, ‘set’, etc) to relatively complex tasks (obtain an SNMP table, a specific column or row of a table, etc). The Java-based SNMP package of AdventNet [AdventNet] has been used as a basis to construct the complex management applications mentioned above. For instance, the ‘get-table’ operation is implemented as a succession of ‘get-next’ requests, until all the values of an SNMP table are retrieved. Currently, only SNMPv1 is supported, mainly due to its simplicity. Although the first release of the SNMP protocol is known to exhibit severe security limitations, that does not represent a problem in this case as SNMP requests/ responses are not transmitted through the network, but locally exchanged.

In a typical scenario, an MA passes an arbitrary number of object OID strings to an SF method, which performs the SNMP query and returns the requested values. These values are subsequently processed, if necessary, by an automatically invoked MA method. The value acquired, either directly by the system or as a result of computation, is passed to the SC subsystem, encrypted and encapsulated into the MA’s state.

4.4.1.3.5. Migration Facility Component (MFC)

The role of the MFC is to dispatch upon request an MA object to a specific network device. An MA transfer may be requested either by the MA itself or any other thread that has a reference to the MA object and the MFC component. MA transfers are triggered through calling the `move()` method of the MFC. The latter will invoke the `moveTcp()` or `moveUdp()` method, depending on which transport protocol is used for the MA transfers (see Figure 4.12).

[7] Hashtables can be thought of as key-indexed arrays that enable efficient search of information.

The MFC component is part of both the MAS servers and the manager application. In essence, its operation is the inverse of the MAL's operation. Namely, in the case that TCP is used for MA transfers, a connection is established, the MA state is compressed (`java.util.zip.GZIPOutputStream` class) and then serialised (`java.io.ObjectOutputStream` class) with the resulted byte stream directed into the TCP connection output stream. When the data transfer is completed, the network connection is released. In case that UDP is used as transport protocol, the procedure described above differs only on that the compressed/serialised data are stored in a byte array and packaged within one (or more) UDP packet(s) prior to their transmission (no connection needs to be established/released). Following the MA's migration, the MFC deletes the MA_Info object associated with to the dispatched MA by invoking the corresponding method of the MAR component.

```
/* Moves the MA object to the requested IP address. The returnToManager boolean shows whether the
operational travel of the MA has ended (the MA should return to the manager) while the tcp flag
denotes the transport protocol used for the MA transfers (true for TCP, false for UDP) */
public void move(MA ma, InetAddress addr, boolean returnToManager, boolean tcp);
// Invoked by the move() method should the transport protocol used is TCP
public void moveTcp (MA ma, InetAddress addr, boolean returnToManager);
// Invoked by the move() method should the transport protocol used is UDP
public void moveUdp (MA ma, InetAddress addr, boolean returnToManager);
```

Figure 4.12. The method definitions of the `MigrationFacilityComponent` class

The life cycle of an MA object and its interaction with the visited MAS servers is illustrated in the block diagram of Figure 4.13.

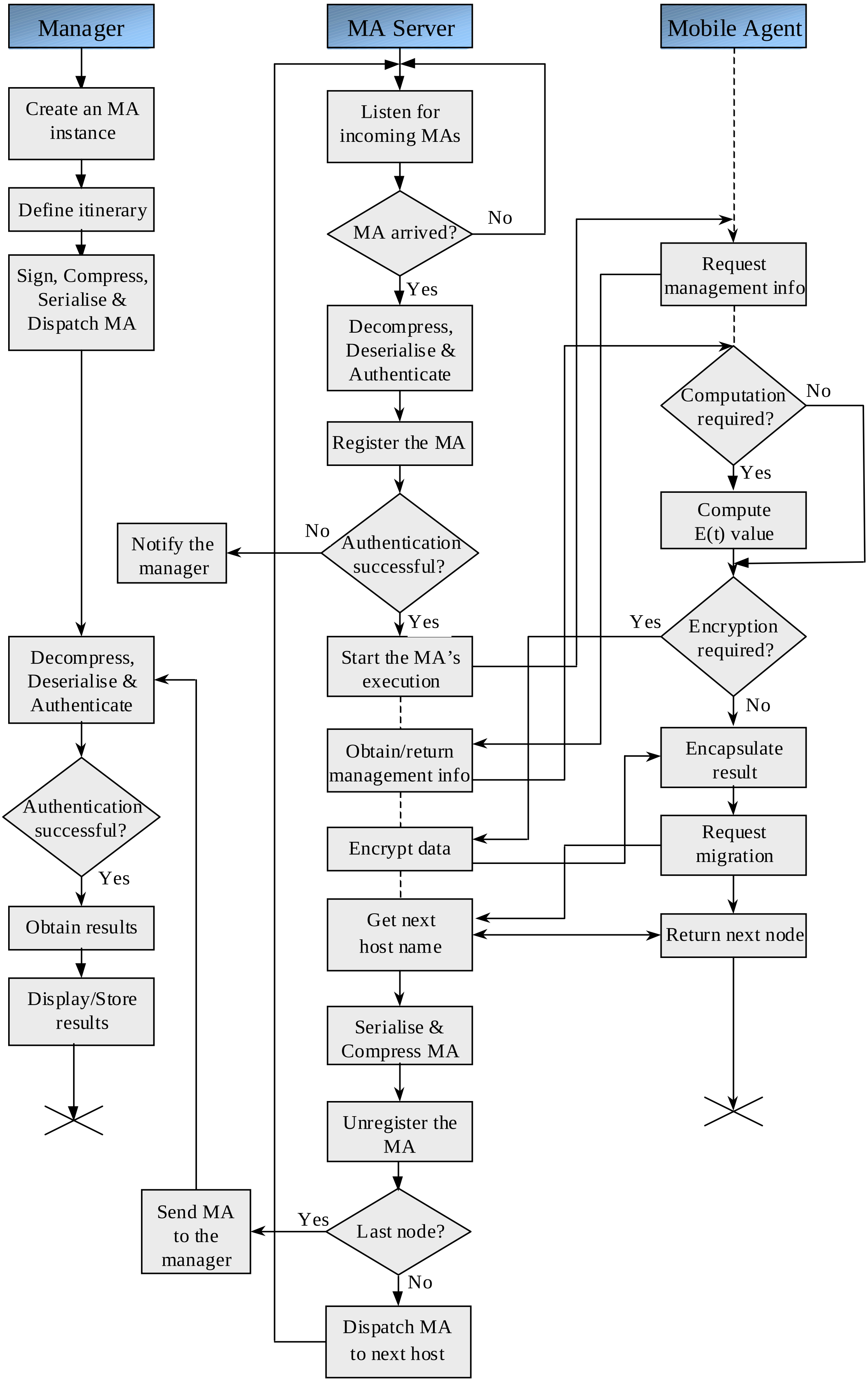


Figure 4.13: Flow diagram of a Mobile Agent's life cycle

4.4.1.3.6. Mobile Agent Server RMI Server

The RMI server is implemented as a separate thread, which can optionally be instantiated by the main MAS thread (this is indicated by the user in the command line that starts the MAS application). The reasons that the RMI server presence is not compulsory is that it does not directly interact with incoming MAs while it represents additional overhead on the local system resources (mainly because of the `rmiregistry` process). The role of the RMI server is to enable remote interaction of the manager with distributed MAS entities and, hence, allow the administrator to obtain reports regarding the CPU and memory load profile of network devices or the number and type of MA objects currently executing on them. It also allows the administrator to perform a number of actions upon the MAs, such as to change their execution status, modify their itinerary, etc (see Figure 4.14).

```
public Vector getResourcesReport (ReportDescription descr) {}  /* Returns a report on the local
                                                                   resources (CPU & memory)  usage */
public Vector getMAsReport () {}   /* Returns a report about the MAs currently executing on
                                      the local device */
public boolean suspendMA (String id) {}  // Suspends the execution of the MA with ID = id
public boolean resumeMA (String id) {}  // Resumes the execution of the MA with ID = id
public boolean activateMA (String id) {}  // Activates the MA with ID = id
public boolean deactivateMA (String id) {}  // De-activates the MA with ID = id
public boolean disposeMA (String id) {}  // Disposes the MA with ID = id
public boolean disposeAll () {}  // Disposes all running MAs
public boolean modifyItininerary (String id, Vector itinerary) {}      /* Modifies the itinerary of the MA
                                                                          with  ID = id. */
```

Figure 4.14. The methods of the MAS's RMI server class

4.4.1.3.7. Resource Inspection Application (RIA)

The RIA is an application developed in C programming language, which runs outside the boundary of the MAS server. Its purpose is to monitor the usage of local resources in terms of CPU and memory load. RIA is linked to the MAS application via the Java Native Interface (JNI) that enables the inter-operation of Java applications with programs written in other languages (see Section 2.7.1.1). That is, JNI allows RIA to invoke certain MAS methods and vice versa. The motivation behind RIA's development has been to allow the administrator but primarily the MA objects to obtain information regarding network devices load. This information is exploited (see Section 6.3.6) to provide even distribution of processing and memory load among distributed agent servers.

4.4.1.3.8. Network Discovery Daemon (NDD)

The NDD is another thread instantiated by the main MAS thread, whose purpose is to discover the station where the manager application runs, while also making the local host 'visible' to the manager. Its operation is essentially very similar to the Network Discovery thread of the manager application (see Section 4.4.1.1). In particular, upon startup the NDD parses a text file including a list of the IP addresses of devices where manager applications are likely to run and ‘pings’ the corresponding hosts, checking whether there is an active manager thereon. When the manager receives the message, it appends the MAS’s hosting device to its ‘discovered’ list and notifies the NDD accordingly.

4.4.1.3.9. Class Loader Daemon (CLD)

The CLD is a daemon controlled by the MAS thread, whose role is to wait for MA class definitions sent by the manager application at the time that a new management service is introduced. The received MA classes are stored in a designated place (directory). CLD’s role is discussed in the following section, while the class loading mechanism used by our framework is described in detail in Section 4.4.3.

4.4.1.4. Mobile Agent Generator (MAG)

The MAG, is essentially a factory for constructing customised MAs. In the context of this thesis, generated MAs are designed to poll static management agents according to certain operational function requirements. A GUI, dedicated to the MAG tool, allows the operator to:

- assign a name to the MA;
- specify the generic type of service this MA is intended to carry out;
- define the MA’s functional requirements, i.e. determine its operational behaviour;
- set the polling frequency to determine how often instances of the constructed MA will be launched;
- specify the transport protocol (either TCP or UDP) to be utilised for the MA transfers;
- determine whether the data collected by the MA are to be encrypted and the MA itself authenticated;
- specify the class of network devices to be polled;
- optionally, define the MA’s itinerary, i.e. the order in which the MA will visit the managed devices.

Options for editing the attributes, deleting or updating an existing MA instance are also available. Snapshots of the MAG GUI are presented in Chapter 7.

4.4.1.4.1. MAG tool operation

The MAG uses a skeleton Java source code with empty slots filled with the user-specified MA’s properties. The Java code created is then compiled through an invocation of the `compile()` method of `sun.tools.javac.Main` class (this class is offered by Sun, although not included in the standard JDK library) [SunTools]. The generated Java bytecode is subsequently compressed and transferred (multicasted) through TCP connections to all operating agent hosts. The MA's properties are compared, prior to its construction, against those of the existing MA classes to ensure that there is no other with the same functionality; in case such a class exists, the user is notified through a message.

On the agent side, the CLD receives and decompresses the transmitted bytecode, validates the included Java class and stores it in a designated space. MAG's functionality is illustrated in the block diagram of Figure 4.15.

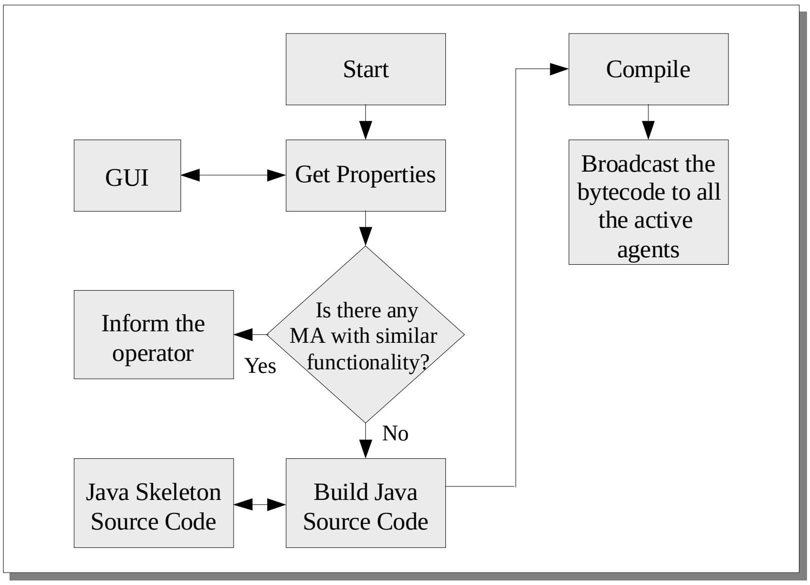
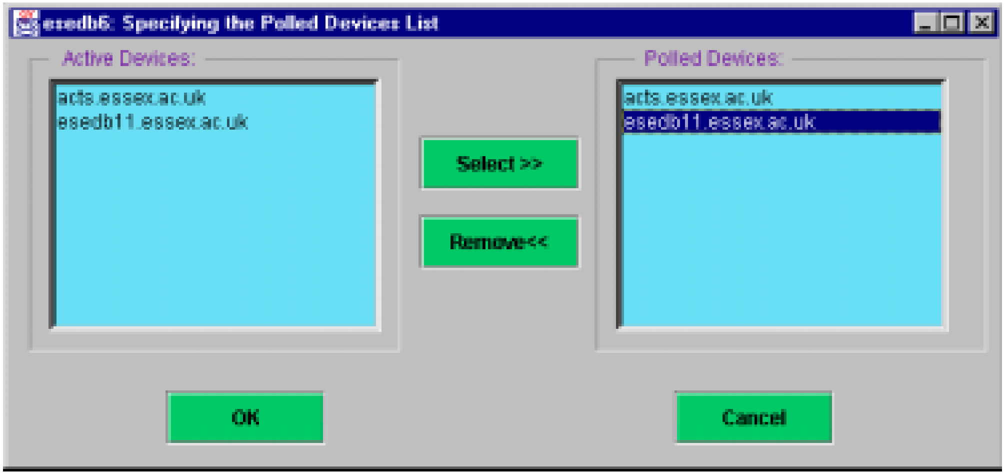

Start
Compile
GUI
Get Properties
Broadcast the bytecode to all the active agents
Inform the operator
Is there any MA with similar functionality?
Yes
No
Java Skeleton Source Code
Build Java Source Code

Figure 4.15: Mobile Agents Generator functional diagram

It should be emphasised that the transfer of the MA bytecode is performed only *once*, at the MA construction time. From that point onwards the transfer of persistent state, obtained from serialising the MA instance, is sufficient for MAS entities to recognise the incoming MA and recover its state. In contrast, most available MAPs apply a policy that requires the transfer of both the MA's bytecode and persistent state, resulting in higher demand on network resources. To illustrate, in our implementation the ratio (bytecode size):(state size) typically lies in the range 10:1 to 15:1.

It should be emphasised that MA code is not 'blindly' multicasted. For instance, in case that a monitoring task is intended to be performed upon a limited set of devices, it would not be meaningful to broadcast the bytecode of the MA to perform this task to every managed device, as that would create unnecessary overhead. Hence, the administrator can specify the set of devices to be monitored (through the GUI shown in Figure 4.16), with MA code being exclusively sent to them.



Active Devices: acts.essex.ac.uk, esedb11.essex.ac.uk

Polled Devices: acts.essex.ac.uk, esedb11.essex.ac.uk

Figure 4.16. Specifying the devices to be monitored.

The functionality of MAs created by the MAG may only be an extension of limited generic service types. These types are designed as sub-classes of the MA ‘super-class’, specifying general patterns of MA-based NSM tasks. MAs constructed by the MAG tool extend one of these sub-classes, refining their functionality and defining service-specialised MAs. Examples of available generic NSM service types will be given in Chapter 7, that describes several MA-based network monitoring applications. An alternative approach for structuring MA-based applications would be to keep a unique MA structure (class), whose functionality would span a wide range of management operations. However, such an approach would result in the unnecessary transfer of dynamically growing code, much of which may only be occasionally executed.

4.4.1.4.2. Advantages of using the MAG tool

The focus of current research on MA-based management is on the development of execution platforms and applications for MAs. However, methodologies for building agents have received little attention. Creating MAs can be tedious and susceptible to errors and also requires programming skills and detailed knowledge of the MAP design. In particular, introducing a new management operation would comprise the following steps: (a) design and write the source code that defines the MA functionality; (b) compile the source code (significant amount of time is wasted for coding and debugging, where programming expertise is a prerequisite); (c) possibly modify and re-compile the core MAP component classes in order to make possible the instantiation of the new MA class; (d) possibly reboot the manager application to allow modifications to take effect; (e) reboot active MAS servers (unless their class loader is able to recognise that the MA classes have been modified).

To ‘condense’ this perplexing procedure in one sentence, we quote from [KIM98]: “If you want another management function, just write another mobile code and ship it again through the network”.

Therefore, the use of the MAG tool brings forth a number of advantages:

- The MAG ensures that the framework remains sufficiently flexible by enabling on-the-fly construction of service-oriented agents, without the need for reconfiguration, re-installation or re-instantiation of either the manager or the agent applications. The MAG functionality can easily be extended so as to cover a wider range of management tasks.
- Ease in introducing new management tasks in a user-friendly manner. The productivity of the development process is increased through reducing the time needed to develop an MA and improving reliability as a result of reusing proven components.
- The generation of the MA bytecode and its distribution among distributed managed hosts is automated and transparent to the user.
- The network administrator does not need to be aware of the implementation details behind the introduced management service or the management framework, nor to have programming experience.

Similar work has been recently reported in [GSC99], which describes the design and implementation of an agent construction toolkit, the AgentBean Development Kit (ADK), which is an extension of the Sun's Bean Development Kit (BDK).

4.4.2. Fault Tolerance: Tolerating Node Failures

A key aspect of our framework is its fault tolerance features. A MAP should be able to survive network and systems failures to secure MA migrations. In particular, the following two fault scenarios have been investigated:

Scenario 1: An MA should adapt to unexpected situations, such as the failure of a host MAL thread. In this case, if TCP is used for MA transfers, the TCP connection establishment fails (Figure 4.17a). The MA’s *onFailMigration*() method is then automatically invoked to record the unreachable host’s name into the MA’s *problem folder* and retrieve the next destination host from the itinerary vector (Figure 4.17b). The MA will then migrate to this host (Figure 4.17c). When returning to the manager host, the MA reports the failed devices to the manager application, which in turn will take any necessary regenerative actions.

If UDP is the transport protocol choice, the detection of a failed device is more complex as the UDP datagram carrying the MA’s state would simply be lost. A way to get around this problem would be to enforce the destination host to issue an acknowledgement when successfully receiving an incoming MA. In case of a fault, the acknowledgement would never reach the originating node and after a given time interval the MA image would be transmitted to its next destination. However, that approach (proposed in [SAH98]) would create additional traffic load, counterbalancing the benefit of UDP lightweight nature.

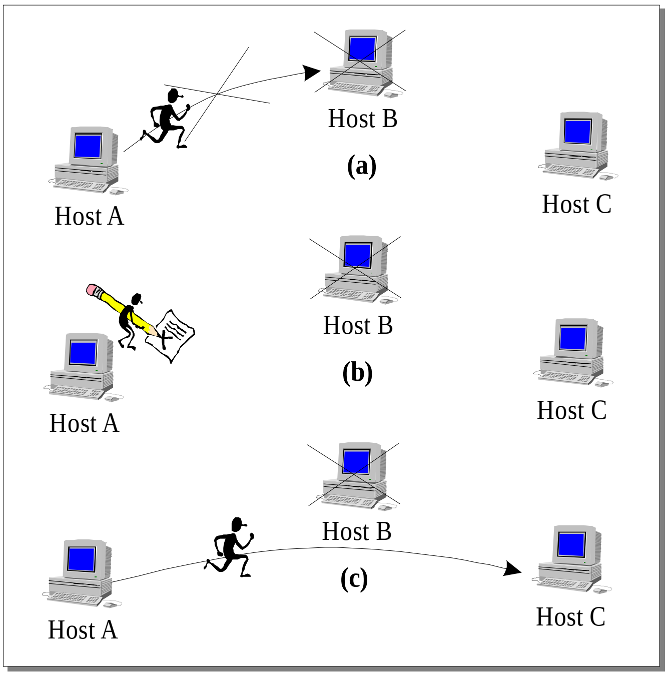


Figure 4.17. MA reaction to the detection of a failed MAL thread

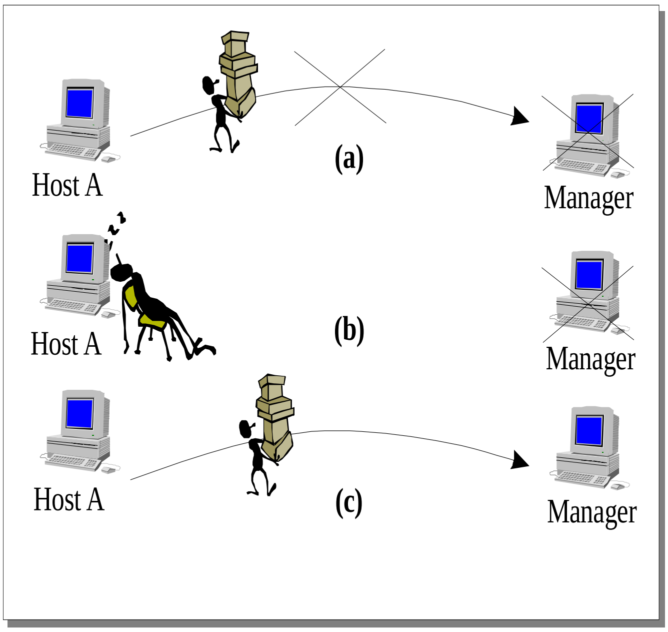


Figure 4.18. The specific case of MAL thread failure on the manager host

Scenario 2: A second scenario would be a fault in the MAL thread of the manager host itself (Figure 4.18a), which represents a special case. Since loss of management data carried by an MA should be avoided, an alternative approach to scenario 1 is employed. Upon detecting a fault, the MA ‘sleeps’ for a given interval (Figure 4.18b) and then resumes execution to retry the connection. If the manager application recovers before a pre-determined number of retries elapses, the MA is transferred (Figure 4.18c), otherwise it is disposed of.

The two algorithms described above have been examined in [BRU01] (they are termed “Skip” and “Wait-and-Retry” respectively); the paper evaluates the algorithms through analytical models and simulation results. It is noted that the reliable operation of the described fault tolerance mechanism has been verified by creating ‘virtual’ failures. In particular, fault scenario 1 has been ‘simulated’ by instructing MAs to visit hosts that do not run a MAS server, while surviving fault scenario 2 has been confirmed by shutting down the manager application and restarting it, allowing travelling MAs to deliver their results.

4.4.3. Class Loading Mechanism

When an MA object arrives at a destination host, its bytecode is retrieved from the network stream, stored in the local disk and cashed within a hashtable of the default JVM’s CL (`java.util.Hashtable`). Since memory access is typically much faster than disk access, the hashtable minimises the time needed to load the MA bytecode when another MA object of the same class visits the same host in the future (it is assumed that a class loaded once, is likely to be loaded again).

The default CL works fine as long as MAs’ definitions remain unchanged; it is not able though to distinguish between two different versions of the same class, due to a limitation of the Java language that does not allow to define two classes with the same name within the same CL namespace [VEN98]. Hence, should an updated version of an MA visits a NE where the old class is already stored in the cache, the CL will load the cashed version, even if the updated class has been sent. This causes a problem when MA objects are being deserialised, as the serialised data of the new class are incompatible with those of the old class version. Most MAPs fail to cope with this MAs ‘*versioning*’ problem and, as a result, the user is forced to ‘reboot’ all the MAS servers where the old version of the updated MA class is already stored to allow modifications to take effect. This solution is certainly not desired, especially for large-scale distributed systems where the hosting devices of MAS servers are geographically dispersed. The fact that MA classes carrying out management operations typically need to be modified/updated on a frequent basis, makes the need for developing a CL that efficiently deals with the versioning problem more pressing. The intelligent and lightweight class loading mechanism used by our framework represents one of its most distinctive features.

Among other requirements, the customised CL should be able to distinguish between two different versions of the same class, i.e. to force a class definition to be updated as it evolves, while reducing the communication cost as well as moderating the usage of NEs computational resources. The Aglets MAP is the only known publicly available platform that deals with the versioning problem; it allocates different CLs to different sets of classes, with a new CL created for each updated version of an MA class [LAN98]. The distinction between the updated and older versions of an MA is achieved by sending information about the names and versions of classes along with the classes' bytecode. A similar approach is adopted in [ISM99] where a CL object is created for each MA class. However, that creates additional network overhead (class version information transferred on every MA migration), while frequent MA classes updates will trigger the creation of many CLs, with increased memory requirements.

A different approach is used in the MAP introduced in this thesis. In particular, a *customised* Mobile Agent ClassLoader (MACL) class has been designed, which extends the default (`java.lang.ClassLoader`) CL class and overrides its `loadClass()` method. The CLD thread of the MAS server maintains a list of the classes that have been already received; thus, when a class definition is received, the CLD checks the received classes list and should the same class has been received in the past, the MACL is forced to load the class definition from the local disk rather than from the cache.

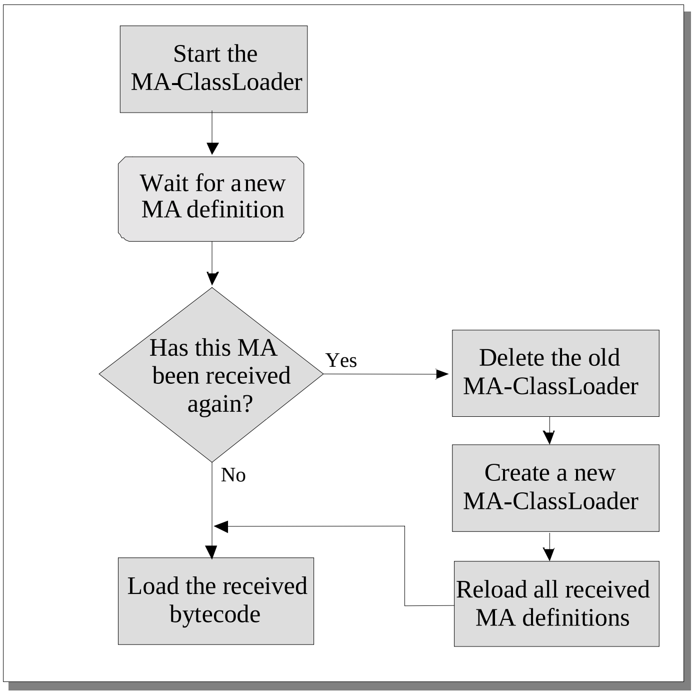

Start the MA-ClassLoader
Wait for a new MA definition
Has this MA been received again?
Yes
No
Delete the old MA-ClassLoader
Create a new MA-ClassLoader
Reload all received MA definitions
Load the received bytecode

Figure 4.19. Class Loading mechanism block diagram

The main advantage of the proposed class loading mechanism is that a *single* MACL object is used, leading to minimal use of local resources. This object is "replaced", i.e. the old MACL object is garbage collected, with a new one taking its place whenever one of the existing MA definitions is modified. The new MACL object then loads the bytecode (definitions) of all the existing MA classes in order to minimise the time needed to load these MAs when they first visit the NE (otherwise the bytecode would be loaded from the disk upon the MAs reception leading to increased delay). The "replacement" technique was necessary, as a single CL does not load the bytecode of a given class twice. The operation of the MACL is graphically depicted in the block diagram of Figure 4.19.

In order to force the MAL thread to load the incoming MA object using the customised MACL (and not the JVM's default CL), a new class (`MCode.MasObjectInputStream)` extending the `java.io.ObjectInputStream` class and overriding its `resolveClass()` method, has been implemented. Thus, by using the `resolveClass()` method of the `MAsObjectInputStream` class, the MAL thread instructs the customised MACL to load the latest version of the incoming MA definition.

The MACL design also caters for the improbable case that the definition of a visiting MA object is not found in the local disk space (potentially because the local device was not originally specified by the administrator among the ones to be monitored by the MA). In this case a `java.lang.ClassNotFoundException` is thrown, the MACL contacts (through RMI) the MCR thread of the manager application and 'downloads' the requested class file.

4.5. Quantitative Evaluation & Assessment

A critical omission frequently noticed in research papers introducing new MA-based frameworks for NSM is the analytical evaluation of the performance issues arising when implementations of the corresponding models are used in real networking environments. With very few exceptions (e.g. [LIO99] that proposes detailed mathematical models for comparative analysis of various MA-based management solutions depending on the MA organisation and deployment patterns), most of these works claim improved performance over alternative approaches, without providing any proofs. There are also several works (e.g. [SAH98, ELD99, ZAP99]) that present simplistic quantitative evaluations, oriented to their respective presented prototypes.

Reference [BAL98] comprises an interesting theoretical investigation of the three Mobile Code paradigms (COD, REV and MAs). A general performance comparison among these approaches is also provided, that may serve as a reference point for related quantitative evaluations. In this section, we undertake a preliminary evaluation, in terms of bandwidth usage and response time, comparing the performance of SNMP against that of our framework. The results of this theoretical investigation will be contrasted to the empirical results presented in Section 4.6.

4.5.1. Response Time Evaluation

In this section, we model the response time measured when performing SNMP or MA-based polling operations. First, let us consider SNMP-based polling (see Figure 4.20a). Assuming that SNMP requests are broadcasted to all the polled devices with the responses collected in parallel (*asynchronous* polling), the overall time for centralised polling, will be:

$$T_{SNMP,a} = max\left(2\left(t_{del,i} + t_p\right) + t_a\right) + N * t_{comp}\text{, for } i = 1..N \tag{4-1}$$

where N accounts for the number of agents being polled, $t_{\mathrm{del,i}}$ is the average network latency between the manager and the i^{th} agent, t_{p} the processing time required for handling a single request/response message, t_a the time needed to access the system resources, and t_{comp} the computation time needed to process the collected data. The overall response time is determined by the maximum individual time needed to poll any of the polled devices (hence the *max* operator). Clearly, t_p and t_a depend on the number v of MIB values obtained from each host:

$$t_p = t'_p + (v-1)\Delta t_p \quad and \quad t_a = t'_a + (v-1)\Delta t_a \tag{4-2}$$

where t'_p and t'_a are the time needed to process the packet or contact the SNMP agent respectively, when there is a single OID in the request packet varbind list; Δt_p and Δt_a are the extra time needed to process and retrieve each extra requested object, respectively.

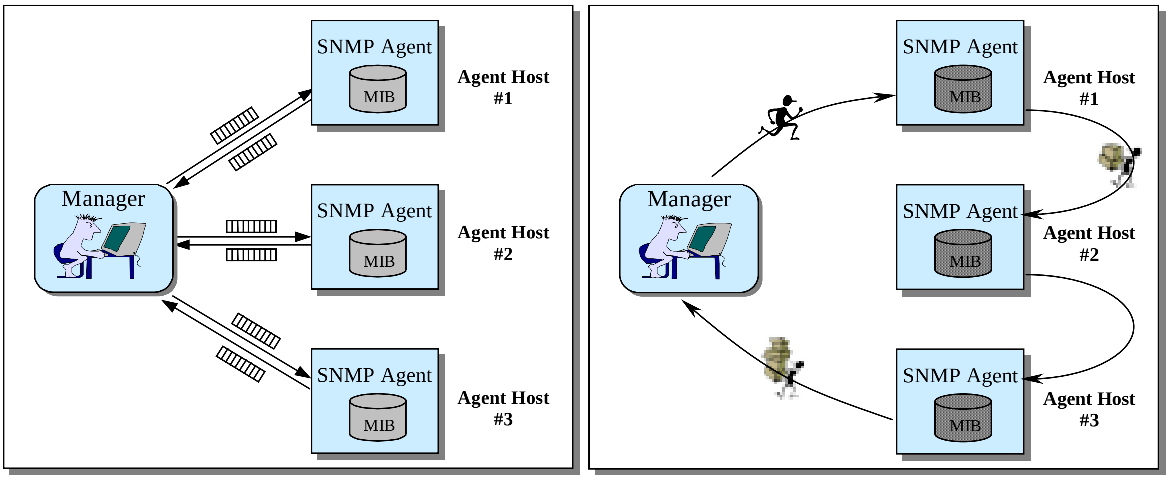


Figure 4.20: Centralised (SNMP-based) vs. MA-based polling

In case that the manager polls the agents sequentially, waiting for the response before sending the request to the next agent (*synchronous* polling), the overall response time is given by:

$$T_{SNMP,s} = \sum_{i=1}^{N} \left(2\left(t_{del,i} + t_p\right) + t_a + t_{comp}\right) \tag{4-3}$$

Regarding MA-based polling, we consider the simplest case where a multi-hop MA sequentially visits every network device (Figure 4.20b). In this case, the response time is modelled as follows:

$$T_{MAs} = t_{cr} + (N+1) * (t_{del} + t_{s/d}) + N * (t_a + t_{comp}) \tag{4-4}$$

where t_{cr} is the time needed to create and instantiate an MA object, t_{del} the average network latency between the manager and the agent or any pair of agents, and $t_{s/d}$ the time taken to serialise/de-serialise an MA. T_{MAs} is defined as the time needed to create an MA object, plus the total transition time to visit each device and return to the manager (hence the factor N+1), plus the time needed to access and process the management information locally. Appendix B proposes programming techniques to accelerate MA migrations, thereby reducing MA-based polling response time.

Clearly, although useful, the modelling of response time for centralised and distributed polling is not enough to provide a clear understanding on how their corresponding response times scale. It is essential to conduct experiments incorporating response time measurements to evaluate the weight of the time factors mentioned above. The results of such experiments are presented in Section 4.6.

4.5.2. Network Overhead Evaluation

As evidenced in [FUG98], a key factor affecting performance is the overhead induced by transport layer protocols. In particular, TCP is more reliable, yet, traffic intensive resulting from its connection-oriented nature. In contrast, connectionless UDP sacrifices reliability in favour of a lightweight communication mechanism [TAN96]. In our implementation, the choice of the transport protocol is left to the network operator.

Examining SNMP-based polling, if S_{req} and S_{res} are the sizes of the request and response packet respectively and O_T is the overhead imposed by the transport protocol (UDP in the case of SNMP), the polling of N devices for p Polling Intervals (PI) would result in wasted bandwidth of:

$$B_{SNMP} = (S_{req} + S_{res} + 2 * O_T) * p * N \tag{4-5}$$

Given that the sizes of the request and the response packets are almost identical (see Section 2.4.1) and assuming that v MIB variables are obtained from each host, the previous equation becomes:

$$B_{SNMP} = ((2 * S_{req}) + (v-1) * \Delta S_{req} + 2 * O_T) * p * N \tag{4-6}$$

where every extra value included in the SNMP response packet's varbind list represents an additional overhead of ΔS_{req} bytes, on average. We also assume that the number *v* of retrieved MIB variables is such that allows the varbind list to be packaged within a single datagram without exceeding the SNMP packet's length limit.

It should be emphasised that the topological structure of the managed network is not taken into account. In other words, it is assumed that the cost of transmitting a certain amount of information from a device to another is independent of their location within the network. The effect of this parameter on the evaluation of network overhead will be investigated in Chapter 6.

Regarding MA-based polling, in overall MA state is transferred *N*+1 times in every PI, including its return back to the manager station. At each point of contact, the MA retrieves a certain amount of management information *b*, subsequently processed by applying a filtering operation. The polling cost highly depends on the increment rate of the MAs state size, which in turn is a function of "*selectivity*" σ ($0 \le \sigma \le 1$), a metric defined in [LIO98] as the proportion of data *delivered* to that *acquired* from each host. For high selectivity values (the major part of the obtained data being filtered at the source) the MAs state size practically remains constant, otherwise the state rapidly grows. Thus, an MA's state size at its *i*th hop is given by:

$$ST_i = ST_0 + (\sigma * b) * i \qquad (4\text{-}7)$$

where ST_i represents the compressed state size of an MA when migrating from the *i*th host (ST_0 is the initial state size). The network overhead imposed by MA-based polling is quantified by:

$$B_{MAs} = \underbrace{N * (C + O_T)}_{Bytecode\ transfers} + p * \underbrace{\sum_{i=0}^{N} (ST_i + O_T)}_{State\ transfers} \qquad (4\text{-}8)$$

The first term of Eqn. (4-8) describes the overhead imposed when multicasting the MA code to all active MAS servers, whilst the second represents the bandwidth consumed by the MA state transfers between the manager and the polled devices. Post initialisation only the MA state is transferred as the corresponding bytecode is multicasted only once, at the MA's construction time; hence, MA state transfers dominate on the overall overhead when considering long-term monitoring tasks. It is therefore important to apply optimisations so as to minimise the persistent state of MAs, thereby reducing the associated management cost. Such optimisations are proposed in Appendix B.

MAs state size increases exponentially as a function of the hop count (number of visited polled devices). This is demonstrated in Figure 4.21 that shows how the management cost per

PI varies as a function of the network size for various selectivity values (it is assumed that the initial MA state size is 400 bytes, the amount of retrieved data is 1000 bytes and that *no* MA state compression is performed). On the contrary, management cost increases linearly with selectivity (see Figure 4.22). The bytecode transfer and the transport protocol overhead have not been taken into account when drawing these figures, which basically illustrate the dependence of state transfers overhead ($\sum_{i=0}^{N} ST_i$) on the network size N and selectivity σ respectively.

Interestingly, following the deployment of MA bytecode, MA-based approach results in a number of message network transfers per polling interval which is almost the half of the ones generated by SNMP (the ratio is (N+1):2N). This observation is of particular importance when many managed devices are located on network segments remote to the manager station. In such case, SNMP will create considerable traffic through the inter-connecting links, whereas an MA object following an ‘intelligent’ itinerary plan would minimise their utilisation through consecutively visiting the hosts residing on individual remote management domains.

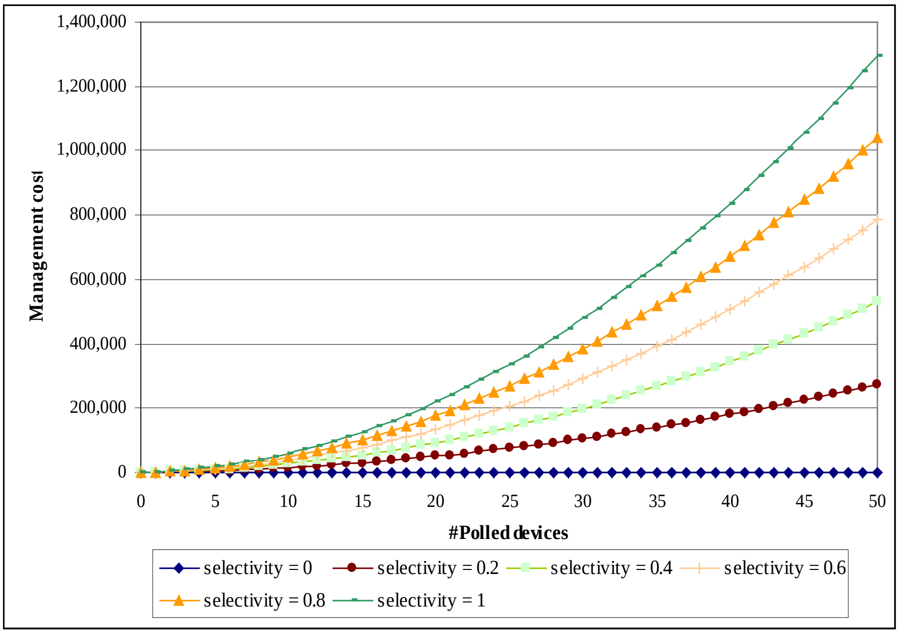


Figure 4.21. Management cost of MA-based polling as a function of the network size for various selectivity values.

The findings of this simple evaluation suggest that MA-based polling performs better than its centralised counterpart only in case that: (a) the volume of retrieved management data is low, (b) the selectivity values are small, (c) the number of polled devices visited by a single MA object is limited. As expected, in any other case MA-based polling does not scale well and may even perform worse than the centralised model. However, the most important conclusion deduced from this evaluation is the exponential growth of MA state size with the number of visited NEs.

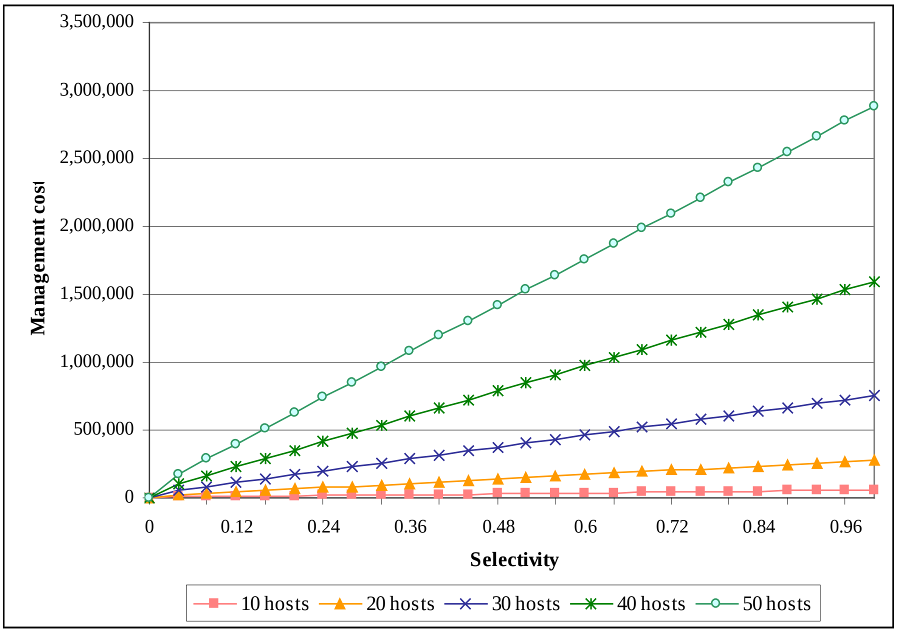


Figure 4.22. Management cost of MA-based polling as a function of the selectivity for various network sizes.

4.6. EXPERIMENTAL WORK

To thoroughly evaluate the proposed framework's performance, a number of experiments have been conducted. In particular, the MA-based solution is compared against both SNMP and a static distributed objects approach based on the Java RMI. This comparison follows two directions: measuring the response time and the network overhead related to management operations.

Response time: The time needed to complete a number of network performance management operations is of critical importance when considering real-time management operations. It is therefore important to experimentally evaluate the response time of MA-based applications and compare it against distributed object approaches. The effect of a number of polling parameters on the overall time also needs to be elucidated. `Utilities.Timer` class has been created to count the timestamps; it includes a number of methods that allow an application to instantiate, start and stop a timer, obtain statistics such as average, standard deviations (SD), etc, print these statistics in text files, etc. The `Timer` class uses the `currentTimeMillis()` method of `java.lang.System` class, which returns the current time in the precision of milliseconds.

The response time experiments presented in this section, start with a performance comparison of MAs against a representative distributed objects technology (DOT), Java RMI.

In particular, a simple experiment is conducted where a string of variable length is transferred from a remote device to the manager station either through an MA object or through an RMI call. A second set of measurements deals with the overall time needed for a multi-hop MA to visit a group of nodes and obtain a certain amount of information from each of them. Additional timing experiments are described in Appendix C. These experiments aim at providing a better understanding of the response time measured for MA migrations; the objective is to investigate how the overall time is distributed among the individual phases of an MA object migration and what is the effect of a number of factors such as the transport protocol used and MA state size.

Network overhead: The traffic generated by the proposed management applications needs also to be measured and compared against alternative approaches. The *WinDump* network analyser [Windump], developed at the Politecnico di Torino, has been used for the network overhead experiments. This program, which represents a graphical version of the well-known *tcpdump* analyser, is capable of capturing packets transferred over a LAN and decoding their contents, providing a useful insight of ongoing network interactions.

In particular, the traffic incurred through MA-based operations is measured and compared against the traffic generated by Java-RMI invocations. The effect that data compression and the transport protocol used for MA transfers have on the measured traffic volume is also weighted.

Regarding the *experimental testbed*, all these measurements have been taken over a lightly loaded Ethernet in the role of the management network, using ten different machines with the following specification: Windows NT operating system, Pentium III (450 MHz) processor and 128MB of memory. The response time experiments were conducted during off-peak period hours, in order to ensure minimum network traffic fluctuations.

4.6.1. Response Time Experiments

4.6.1.1. MAs vs. RMI

This part of the experimental work focuses on comparing MAs against Java RMI, in terms of response time. As far as the MA-based approach is concerned, two different scenarios are examined:

a) An MA object is instantiated by the manager application and sent to a remote device where it obtains and encapsulates an amount of data (a string of characters) and returns back to the manager to deliver the collected data (Figure 4.23a). This type of MA is termed ‘ping-pong’ MA. In the next PI, a new MA is launched and repeats the same procedure.

b) An MA object is instantiated by the manager application and sent to a remote device where it remains permanently. In every PI, this MA (termed the ‘Master’) creates a clone of itself (termed the ‘Slave’). The Slave encapsulates the obtained data and returns back to the manager to deliver the collected information (Figure 4.23b). In the next PI, the same procedure is followed with the creation of a new Slave agent.

The state of migrating MA objects can optionally be compressed before their transmission.

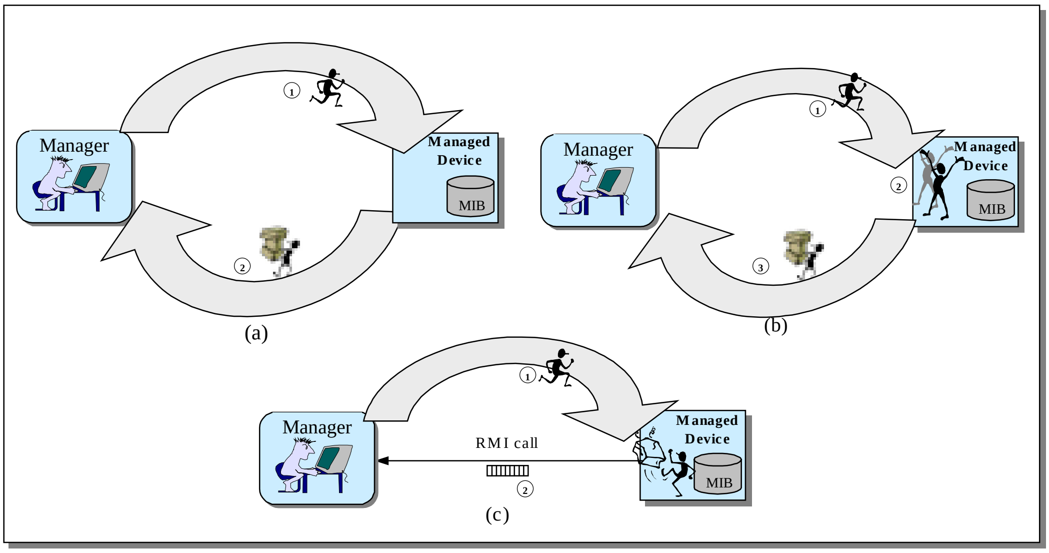


Figure 4.23. Strategies for obtaining information from remote devices: (a) ‘ping-pong’ MA, (b) MA cloning (‘Master-Slave’ scheme), and (c) RMI call.

These two schemes, are compared against a third scheme where an MA object is dispatched by the manager application and remains permanently at a remote site, communicating the obtained results to the manager through RMI calls (Figure 4.23c).

String length (bytes)	MAs (non-compressed, round trip)		MAs (non-compressed, cloning)		MAs (compressed, round trip)		MAs (compressed, cloning)		RMI	
	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
10	84.3	20.2	33.8	6.6	79.9	8.2	31.3	6.0	24.7	14.8
50	84.5	19.0	35.5	5.4	78.7	5.9	26.6	6.9	25.4	14.5
100	83.6	11.0	32.5	8.3	78.1	5.7	29.1	6.4	23.6	7.2
200	86.4	18.0	35.2	10.6	82.9	6.8	26.0	5.3	33.6	96.1
500	88.5	6.5	35.0	5.4	86.9	7.0	32.9	10.8	22.7	6.3
1000	103.9	18.1	42.6	4.6	97.1	13.2	29.3	5.8	24.9	11.3
2000	113.1	17.0	45.6	5.6	108.2	12.7	26.8	7.4	27.7	12.8
5000	147.8	28.6	46.1	7.3	147.7	24.0	35.5	5.0	34.1	11.8
10000	249.2	35.6	62.9	7.5	203.9	20.0	40.4	8.0	43.6	13.5
20000	379.3	33.3	93.1	30.0	306.8	34.0	65.9	20.5	70.8	32.4

Table 4.2. Comparison of MAs vs. RMI-based approaches in terms of response time as a function of the transferred data

The average response times and the corresponding SDs for the three schemes are presented in Table 4.2. The effect of the amount of transferred information on the overall response time is also investigated, hence the length of the string returned to the manager varies from 10 up to 20,000 characters. It is noted that MAs are transferred using the TCP protocol, since it is not feasible to time transfers over UDP. That is because such a measurement would pre-suppose that the clocks of the sending and receiving hosts are perfectly synchronised, which cannot be easily achieved (a way to get around this problem would be to ‘echo’ the UDP message back to the origin host and measure the overall round-trip delay, yet, that measurement would not be very accurate). In contrast, when considering MA transfers over TCP, the sender is able to measure the migration latency, as the completion of the transfer is signified by the receiver’s request to release the connection. In order to perform a direct comparison between MA cloning and RMI-based approaches, the corresponding measurements have been taken at steady state, that is after the code had been shipped to the remote elements, as the latter is performed only once. Code deployment time is not included.

The results displayed in Table 4.2, are also graphically presented in Figure 4.24. As expected, the approach that incorporates a round-trip travel of an MA object scales worse than the scheme that involves remote cloning, as in the latter the MA travels only in one direction. Interestingly, the response times of the MA-based remote cloning approach and the RMI method invocations almost coincide, with RMI exhibiting a slightly worse scaling. Also, in agreement with the conclusions of the previous section, the compression of MA state becomes more attractive as the amount of encapsulated data, i.e. the MA state size, increases.

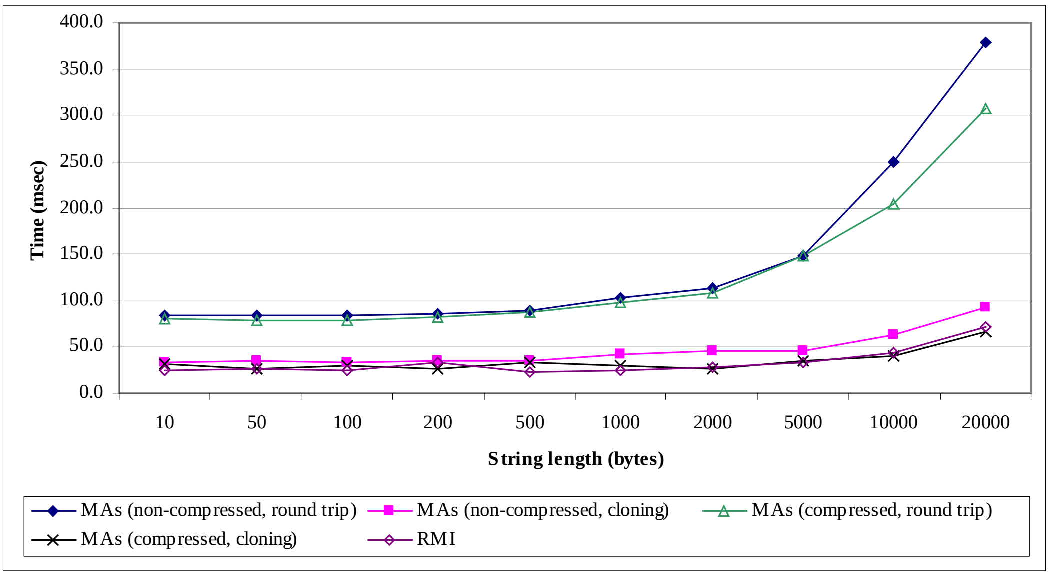


Figure 4.24. Graphical representation of the MAs vs. RMI-based approaches comparison in terms of response time

4.6.1.2. Multi-hop MAs

This set of experiments aims at investigating the effect of several factors on the overall response time of multi-hop MAs, i.e. mobile objects that sequentially visit a number of polled devices, obtain a certain amount of data from each one and return to the manager to deliver their collected data. The evaluated factors are the transport protocol, the network size and the volume of data obtained from each host. The network size varies from 1 to 10 hosts, while the amount of encapsulated data varies from 10 to 100, 500, 1000 and 2000 bytes. No data compression is used in this case to ensure that the increment on the amount of encapsulated data coincides with the MA's state size increment. The results are presented in Table 4.3 and Table 4.4, corresponding to time measurements where MAs are transferred over TCP and UDP, respectively. The same results are graphically illustrated in Figure 4.25 and Figure 4.26.

# Collected bytes (from each host)

# Polled Devices	10		100		500		1000		2000	
	Average	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**
1	84.6	10.3	84.4	18.8	88.8	7.1	89.5	19.4	148.9	328.9
2	85.6	12.2	83.1	9.3	91.4	13.7	100.3	19.1	119.6	20.8
3	89.3	12.4	89.0	16.1	98.5	14.4	113.6	20.2	195.5	41.1
4	108.3	22.1	95.4	15.0	124.8	16.9	159.9	30.3	274.5	29.3
5	108.6	14.9	114.9	20.3	158.8	21.9	220.0	33.5	374.5	31.3
6	138.7	20.5	138.4	13.5	193.8	19.0	301.6	23.6	503.6	38.9
7	149.9	9.5	161.9	18.4	243.5	24.6	383.3	35.4	680.5	74.6
8	173.8	17.4	175.1	20.2	298.3	38.0	475.6	43.2	850.5	67.0
9	185.5	14.5	196.5	14.9	357.5	39.2	620.1	487.3	1098.9	113.5
10	199.3	11.0	230.9	28.4	413.7	41.5	792.8	69.7	1390.1	115.2

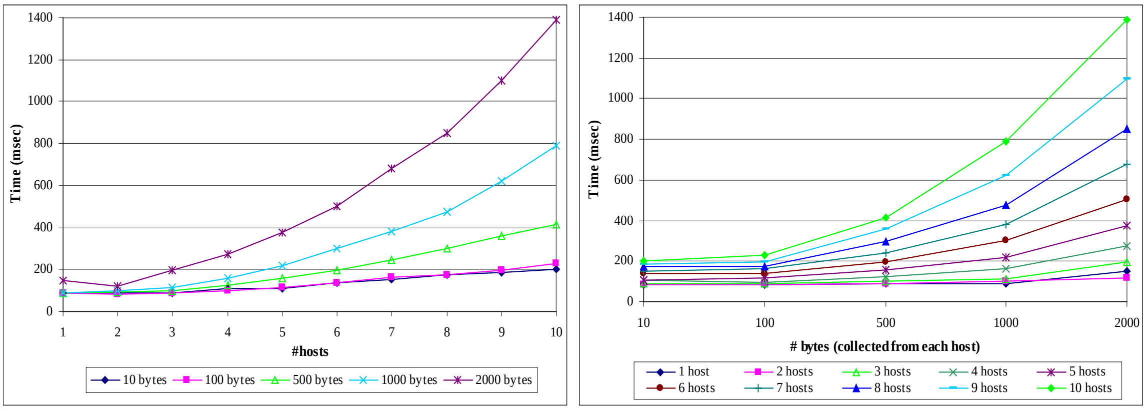
Table 4.3. Average response time and standard deviation for multi-hop MAs for various network sizes and volumes of collected data (TCP protocol used for MA transfers)

# Collected bytes (from each host)

# Polled Devices	10		100		500		1000		2000	
	Average	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**
1	77.6	6.3	78.1	6.5	88.2	14.5	87.3	19.0	94.7	10.8
2	81.2	12.5	86.5	10.8	88.9	9.6	95.7	10.1	113.6	20.9
3	81.1	6.4	84.9	14.3	100.1	17.3	117.6	12.8	185.2	29.7
4	87.3	11.9	95.9	13.8	136.6	15.8	170.0	23.7	300.4	28.0
5	104.7	9.2	117.7	10.7	161.0	12.8	227.0	22.5	399.0	30.0
6	123.2	12.8	137.7	22.4	201.6	19.5	313.9	31.9	531.9	31.9
7	143.1	17.3	162.5	19.1	243.3	21.5	403.1	30.1	701.3	40.2
8	158.7	19.2	183.7	19.1	297.8	27.5	497.9	35.3	858.1	41.0
9	180.2	23.5	210.2	15.9	357.6	30.7	602.7	42.3	1101.1	83.6
10	199.3	24.4	235.6	14.9	414.3	34.6	760.9	63.2	1384.3	108.4

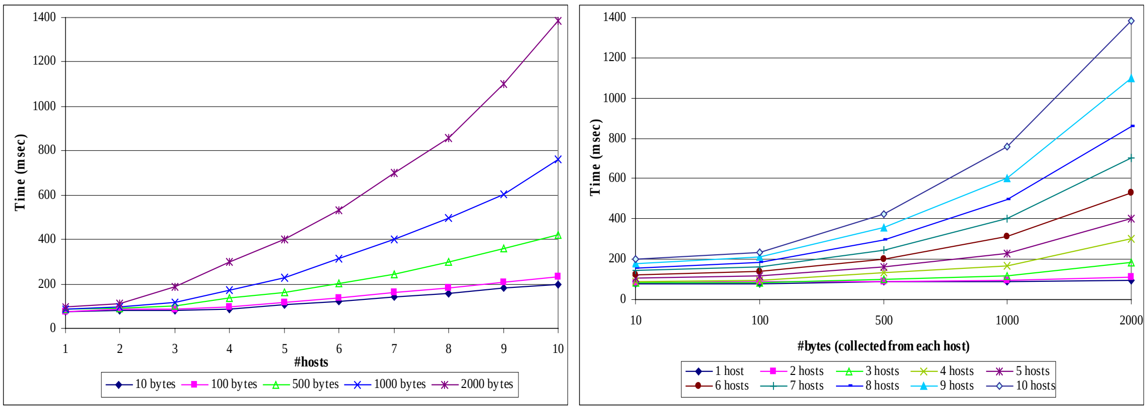
Table 4.4. Average response time and standard deviation for multi-hop MAs for various network sizes and volumes of collected data (UDP protocol used for MA transfers)

The results indicate an exponential growth of the overall response time as the network size increases. Regarding the effect of transport protocol, UDP performs marginally better for small volumes of encapsulated data while the performance of the two protocols converges for higher volumes of data and larger network sizes. The experimental results agree with the conclusions of the quantitative evaluation presented in Section 4.5.2, according to which it is necessary to limit the itinerary length of multi-hop MAs in case that their selectivity is low, i.e. when the volume of collected data at each visited host is high.



(a) (b)

Figure 4.25. Response time for multi-hop MAs as a function of (a) amount of encapsulated information and (b) network size (TCP protocol used for MA transfers)



(a) (b)

Figure 4.26. Response time for multi-hop MAs as a function of (a) amount of encapsulated information and (b) network size (UDP protocol used for MA transfers)

4.6.2. Network Overhead Measurements

An important scalability parameter that signifies the appropriateness of a model on management applications is the volume of data transferred through the network when implementing this model. Although scalability is suggested as the main argument that favours the development of management frameworks employing MAs, very few works (e.g. [BOH00a]) supplement design ideas and implementations with quantitative evaluations or real traffic measurement data. In addition, the decision to adopt MA-based solutions for management applications instead of static DOT-based approaches is rarely justified through systematic performance comparisons.

This section attempts to correct this insufficiency by presenting the results of an experiment that compares the performance of the proposed MA-based framework against Java-RMI, in terms of the network traffic they incur. The experimental testbed is very simple in this case: it comprises a pair of PCs, with the first playing the role of the manager and the second the role of the managed device.

		RMI (random sequence)	**MA (TCP - repetitive sequence)**	**MA (TCP - random sequence)**	**MA (UDP - repetitive sequence)**	**MA (UDP - random sequence)**
	# Captured frames					
String length (chars)	**0**	8	9	9	1	1
	10	8	9	9	1	1
	50	8	9	9	1	1
	100	8	9	9	1	1
	500	8	9	9	1	1
	1000	11	9	9	1	2
	2000	13	9	12	1	3
	5000	19	9	17	1	5
	10000	32	9	25	1	9
	Transferred data on the Transport layer (bytes)					
String length (chars)	**0**	699	842	842	503	503
	10	713	846	858	506	518
	50	776	848	932	508	587
	100	855	849	986	510	664
	500	1527	854	1515	515	1185
	1000	2327	859	2220	520	1838
	2000	3901	865	3490	525	3079
	5000	8609	876	7115	535	6803
	10000	16677	889	13613	550	12957
	Transferred data on the MAC layer (bytes)					
String length (chars)	**0**	843	1004	1004	521	521
	10	857	1008	1020	524	536
	50	920	1010	1094	526	605
	100	999	1011	1148	528	682
	500	1671	1016	1677	533	1203
	1000	2525	1021	2382	538	1874
	2000	4135	1027	3706	543	3133
	5000	8951	1038	7421	553	6893
	10000	17253	1051	14063	568	13119

Table 4.5. Comparison of MAs vs. RMI-based approaches in terms of network overhead: The total number of captured frames and the volume of the transferred data on the transport and MAC layers

First, the manager instantiates and launches an MA object to the managed device. The MA then creates a string (`java.lang.String`), which is either directly returned (passed as a parameter) to the manager through an RMI call (first scenario) or encapsulated into the MA's state and returned to the manager application through a second MA transfer (second scenario). The experiment has been repeated for string lengths that very between 0, 10, 50, 100, 500, 1000, 2000, 5000 and 10000 characters. Focusing our attention on the second scenario, the MA transfers may be performed over TCP or UDP protocol. In addition, in order to assess the effect of data compression, the encapsulated string might either be a repetition of a 10-characters-long substring or composed by randomly generated characters (through using the `random()` method of the `java.lang.Math` class). The use of data compression in the RMI-based approach was not possible, as that would require modification of the code that determines the way that RMI handles input and output streams.

By running the *Windump* network analyser and using appropriate filters, the packets exchanged between the two devices have been captured and analysed. The results of this experiment are shown in Table 4.5, that presents the total number of packets captured during a single transaction, the volume of data transferred through the network on the OSI *transport* layer (including the TCP/UDP headers) as well as on the *MAC* (Medium Access Control) layer (including the Ethernet headers). The latter is also graphically presented in Figure 4.27. Unlike response time experiments, there was *no* need to repeat the measurements since the network overhead could be accurately measured by discriminating it from the background traffic[8].

Clearly, using TCP for transport protocol results in exchanging a much higher number of packets, especially when the size of the actual 'useful' data is small; most of these packets relate to the connections establishment/release phases. On the other hand, UDP, operating on connectionless fashion, does not require the exchange of any control packets and thus results in lower network overhead at the expense of more unreliable data transfers.

Notably, the effect of compression is much more prominent when the encapsulated string represents a repetitive sequence of characters; in such case, there is a high degree of redundancy, which is effectively exploited by the *gzip* compressor to achieve higher compression ratio and diminish network overhead. Compression becomes more effective as the volume of transferred information increases, for instance the compression ratio equals 90% for 50-characters-long string and 99.5% for 10000-characters-long-string. High compression ratios are not, however, feasible when the string is randomly generated. Yet, the latter represents a

[8] *Windump* offers a variety of 'filters' that control the capturing process so that the captured frames are those that conform to certain restrictions, specified by the user.

more realistic scenario, as collected management information is unlikely to exhibit high redundancy.

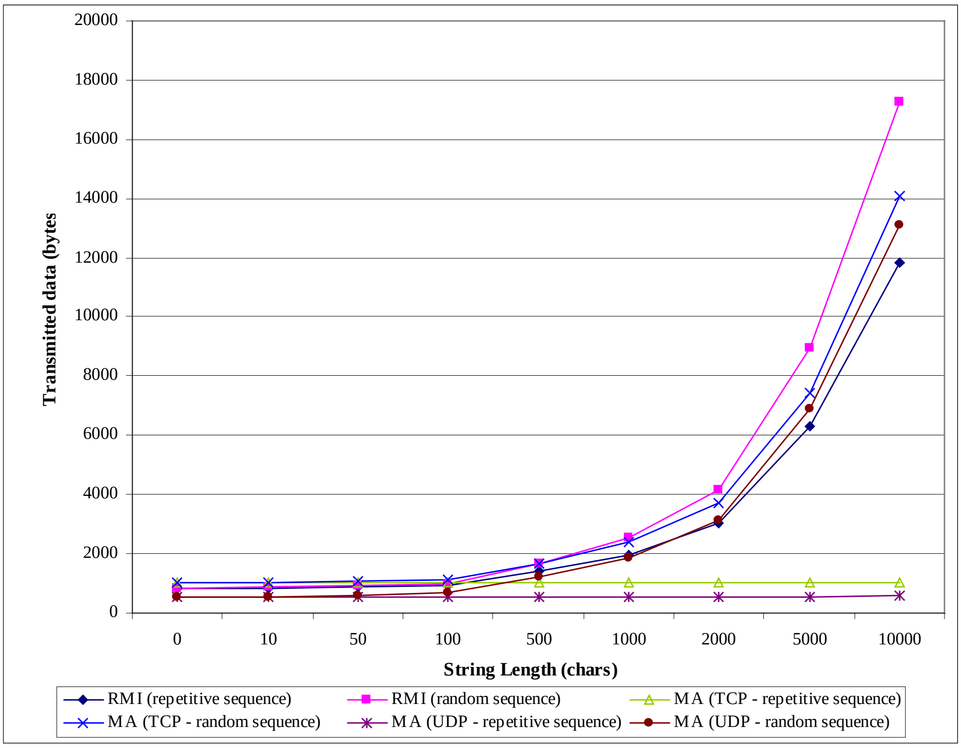


Figure 4.27. Graphical representation of Mobile Agents vs. RMI comparison in terms of network overhead

The most significant outcome of this experiment is that even when encapsulated strings are randomly generated, the performance of MA-based solutions is slightly superior to that of the RMI-based approach, especially as the volume of transferred data increases. The experiment would be more complete if our framework was also compared against CORBA. However, a similar comparison reported in [BOH00a] indicated that CORBA and RMI performances are in the same order of magnitude, with CORBA exhibiting slightly improved scalability. Still, these conclusions may be reconsidered should different implementations of RMI and CORBA are evaluated.

Although the relation of this experiment to management applications might not be evident, it sufficiently demonstrates the competence of the proposed framework against static distributed objects technologies. The results of additional experiments, presented in Chapter 7, will prove the precedence of MA-based solutions over traditional centralised (SNMP-based) models, in terms of network overhead, when considering realistic management scenarios. As a final remark, it should be mentioned that the network overhead incurred in multi-hop MA-based operations is not shown here, but will be examined in Chapter 5 (Section 5.4.2).

4.7. Summary - Concluding Remarks

This chapter comprised a description of the design and implementation of a Java-based MAP intended for use in NSM applications. In order to achieve management orientation the framework has been implemented from scratch, incorporating a variety of features that meet the special requirements of distributed management applications. Hence, the presented infrastructure exploits the benefits associated with MA technology, providing a lightweight design with minimal impact in managed devices, moderate use of network resources, prompt execution of time-sensitive tasks and ease in the introduction of new management operations. The key features of the described framework are summarised in the following:

- Acknowledging the huge installation basis of SNMP within TCP/IP networks, the introduced framework is integrated with an SNMP stack developed in Java, allowing incoming MA objects to locally interact with SNMP agents through the SF component.
- A variety of GUIs offering MIB browsing, data visualisation, performance statistics and high-level management operations have been designed.
- The manager application allows performing both centralised (SNMP) and distributed, MA-based operations upon managed devices.
- MAS servers have been designed so as to have lightweight footprint on system resources and therefore provide execution environments easily installable even on devices with limited storage capacity and processing capability.
- Client/Server interactions between the manager and the managed devices is supported through an RMI server controlled by the manager application and another optionally installed and integrated within MAS modules.
- MAS entities keep an active control over locally executing MA threads through registering and maintaining a reference of the corresponding MA objects. That also allows the administrator to visually profile and remotely control (stop, suspend, resume, etc.) MAs execution through RMI calls.
- A set of security features has been incorporated into the introduced framework. Travelling MA objects are authenticated at each visited host ensuring that they have been originated from a trusted host, their actions are authorised, whilst the data encapsulated into their state can optionally be encrypted. Although not answering all the security concerns raised against MA technology, these features ensure that NEs are reasonably safe against MA attacks and MA objects shielded against malicious hosts.

- In order to improve flexibility and maximise code reuse, MAs are implemented following a flexible hierarchical approach whereby MA classes created by the MAG tool inherit the properties of an MA's 'superclass', which defines the root attributes required to employ mobility characteristics.
- The administrator is given the choice between TCP and UDP as a transport protocol with respect to MA migrations. As shown by the experimental results, the choice of transport protocol may affect both the network overhead and latency of MA-based operations.
- The itineraries of travelling MAs may either by automatically built or manually assigned.
- A graphical tool (MAG) that automates the generation of MA code has been designed facilitating the introduction of new management tasks conforming to generic service patterns.
- A number of fault-tolerance features have been integrated into the framework. These cover the cases where a host included in an MA's itinerary has failed, an interconnecting link has broken or the manager station itself is down.
- A customised MA ClassLoader has been created, which in addition to receiving new instances of management tasks at runtime, is able to distinguish different versions of the same MA class, thus enabling modifications over existing tasks to take instant effect.
- A number of transport protocol parameters have been configured so as to accelerate MA migrations and enable the timely execution of real-time management operations (see Appendix B). Along the same line, MA objects are assigned the highest priority among the processes executing on their hosting devices.
- Based on the observation that the network overhead associated with MA transfers is mainly due to the MA code rather than the MA state information, a lightweight code distribution scheme has been implemented. In particular, MA bytecode is transferred to managed devices at the MAs' creation time, with only MAs state transferred following that.
- To further reduce the impact on network resources, MAs state is compressed (*gziped*) prior to its transfer.

In addition to presenting implementation details, this chapter also included a quantitative evaluation of the introduced framework, providing simple mathematical formulations that quantify the response time and network traffic load associated with centralised and MA-based management. Finally, a variety of experiments have been conducted in order to assess the performance of the introduced framework in practice. The experimental work has been twofold, namely the framework's performance is assessed both in terms of the response time and the generated network traffic involved in management operations.

Experimental results revealed that our framework marginally outperforms RMI both in terms of network overhead and response time when considering simple *data transfers between a pair of hosts* (manager - managed device). Nevertheless, although reducing the number of messages communicated through the network compared to centralised models, the use of MAs for managing multiple NEs should be carefully evaluated. As evidenced in Section 4.6.1.2, the use of a single multi-hop MA that visits all managed devices obtaining a specific amount of data from each of them, represents a *non-scalable solution*, especially for tasks where selectivity is low (i.e. only a small portion of the obtained information is filtered at the source). That conclusion dictates that MAs itinerary length should be limited.

It should also be emphasised that the topological structure of the introduced framework and the distribution of managed devices within the network have not been taken into account in the quantitative evaluation of Section 4.5 or the experimental results, assuming fixed MA migration cost, regardless of the location of sending and receiving devices within the network.

These issues will be addressed in Chapters 5 and 6, where more efficient models that improve the framework's scalability, flexibility and adaptation on changing networking environments are identified.

CHAPTER 5

TWO COMPLEMENTARY POLLING SCHEMES FOR IMPROVING MANAGEMENT SCALABILITY

5.1. INTRODUCTION

The core framework introduced in the previous chapter brings about all the benefits associated with the application of Mobile Agents (MA) in Network & Systems Management (NSM), i.e. dynamically customisable management services, local filtering of management information, etc. However, it suits NSM applications that involve high selectivity values for collected data and relatively small set of devices, concentrated in limited number of management domains. In other words, its scalability is questionable as it represents a 'flat' model, whereby a single MA object is launched from the manager platform and sequentially visits all the managed Network Elements (NE), regardless of the underlying topology (see Figure 5.1). The scalability problem is twofold:

- First, in large networks the *round-trip delay* of the MA greatly increases as the overall travel time depends on the number of hops realised by the MA.
- Second, the *network overhead* imposed by the MA transfers grows exponentially with the network size; the slope of the overhead curve becomes steeper in the case of high selectivity values (see Figure 4.21).

Furthermore, when utilising the flat model approach for the management of a large set of devices, the problem of collected data *inconsistency* arises. In particular, when the administrator requests a 'snapshot' of the managed devices status at a given time, the use of a single multi-hop MA does not represent an appropriate solution. This is due to the non-negligible time intervals between the acquisition of each data sample from every NE. For instance, the data values collected from the first and the last itinerary hosts will refer to distant time instants, affecting the consistency and reliability of extracted statistics.

The 'flat' model has been adopted by the majority of Mobile Agent Platforms (MAP) developed for NSM applications, e.g. [KU97, NIC98, BEL99, PIN99, PUL00b]. All the aforementioned works involve frequent MA transfers, when the collection of management statistics is considered, giving rise to scalability and data consistency concerns for the reasons highlighted above. Hence, these platforms are not appropriate for network monitoring and performance management, which represent the main application areas in this thesis.

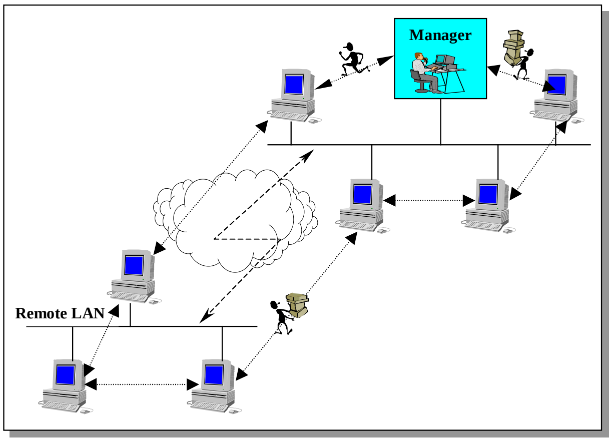


Figure 5.1. 'Flat' MA-based polling

Performance management involves gathering and logging data generated by devices, which may be analysed off-line or in real-time. That process helps in measuring the performance, throughput and availability of network resources. The advantage of analysing the data in real-time is that it allows sophisticated Network Management Systems (NMS) to foresee a possible congestion or failure and take preventive measures before the actual error occurs. On the other hand, collected data may be used to build daily, weekly or monthly graphs/reports to assist the administrator in network planning. In such cases there is no need for real-time NSM data and, hence, an alternative, complementary polling mechanism should be applied.

Therefore, we propose two MA-based polling schemes intended to provide efficient means for obtaining both real-time and off-line management data. The first approach, called *Get 'n' Go* (GnG), is used for the collection of real-time data; the network is partitioned into several domains and a single MA object is assigned to each of them. In every Polling Interval (PI), this MA sequentially visits all NEs within its network domain and obtains the requested information before returning to the manager. The second polling scheme, termed *Go 'n' Stay* (GnS), targets the acquisition of data to be analysed off-line, where the need to obtain data in short time frames is no longer an imperative. Thus, we introduce a method where an MA object is 'multicasted' to all monitored devices; the MA remains there for a number of PIs and collects an equal number of samples. Collected data may be delivered to the manager using various approaches, namely through the broadcasted MA objects themselves, clones of these objects or RMI invocations. The infrastructure described in Chapter 4 has been extended so as to support the introduced polling schemes.

Two papers describing the design and implementation of the two polling schemes have been published in the proceedings of the *3rd International Workshop on Intelligent Agents for Telecommunication Applications* (IATA'99) and the *IEEE Global Communications Conference* (Globecom'99). Full references are given in Appendix A.

This chapter is organised as follows: Section 5.2 describes the design and implementation details of the two introduced polling schemes. Section 5.3 presents a performance analysis regarding the network overhead associated with their use, with experimental results reported in Section 5.4. Section 5.5 discusses the prons and cons of utilising the two polling schemes in practical management applications and Section 5.6 summarises the chapter.

5.2. POLLING SCHEMES : DESIGN & IMPLEMENTATION

5.2.1. Get 'n' Go Polling Scheme

Recently reported NSM-oriented MAPs assume small or medium-sized networks with flat topology structures. Hence, as the number of managed devices grows, the network becomes increasingly unmanageable. This is a consequence of having a single MA responsible for obtaining NSM data from all devices in every PI, giving rise to serious scalability concerns. In addition, the order in which managed nodes are visited is typically arbitrary.

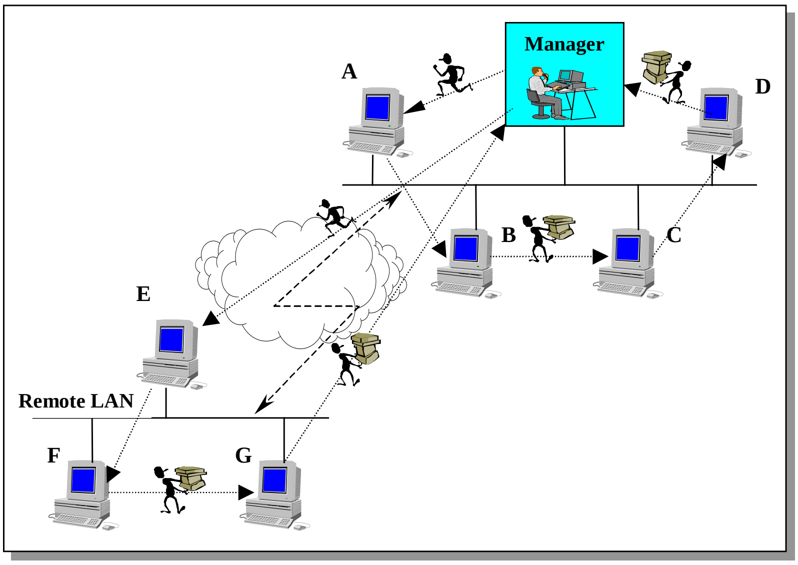
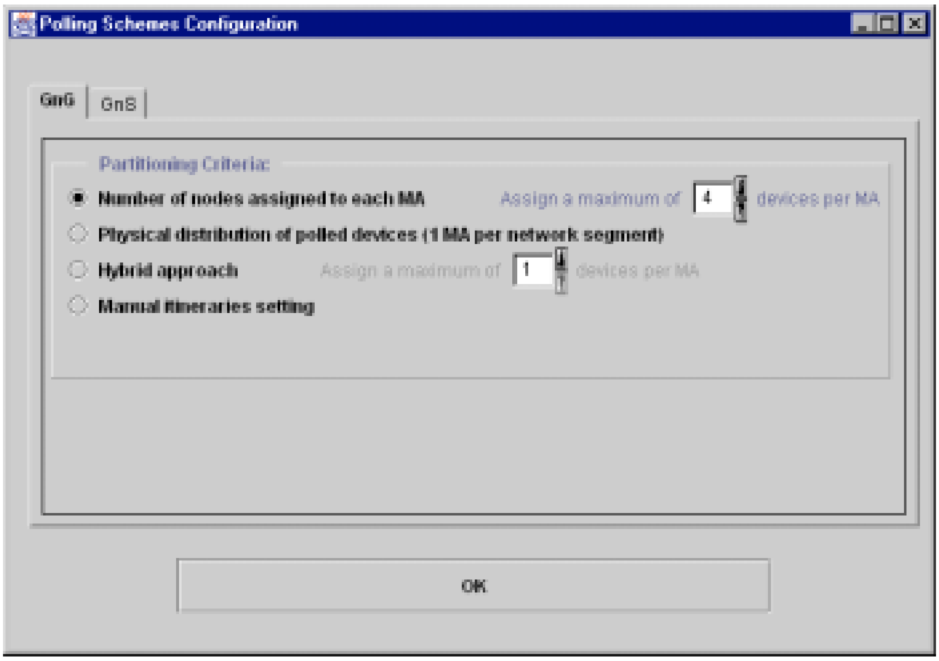


Figure 5.2. The GnG polling scheme

This represents a significant problem when the management of remote LANs is considered, as a travelling MA may have to be transferred several times across expensive and low-bandwidth WAN links during its lifetime. Even worse, when the managed network spans *s* subnets ($s > 1$) connected to the manager site through low-bandwidth WAN links, the overall response time will increase rapidly, as the multi-hop MA will have to traverse a slow link at least $2 \times s$ times.

As previously mentioned, the concept behind GnG polling is to partition the managed network into several logical/physical domains. For instance, in Figure 5.2, an MA object polls the devices of the remote LAN, whereas a second MA is assigned to the network segment local to the manager host. Hence, flat MA-based management can now be thought of as a special case of GnG polling, where the managed network comprises a single management domain.



Partitioning Criteria:

● Number of nodes assigned to each MA Assign a maximum of 4 devices per MA

○ Physical distribution of polled devices (1 MA per network segment)

○ Hybrid approach Assign a maximum of 1 devices per MA

○ Manual itineraries setting

Figure 5.3. GnG polling: Selecting the partitioning criterion

The partitioning criteria are specified by the administrator, during the configuration of the monitoring task parameters. These criteria may be the following:

(a) the number of nodes assigned to each MA, i.e. equal distribution of managed hosts among individual MAs;

(b) the physical distribution of polled devices, i.e. one MA object assigned to each network segment;

(c) a hybrid of these two approaches, i.e. individual MA objects do not visit NEs located on different segments, but there is an upper limit of hosts assigned per MA. In other words, a given network segment may be split in more than one logical domains with every individual domain assigned to a different MA;

(d) manually specified itineraries.

The criterion which is more appropriate for a given management task and its corresponding parameters are selected through the GUI shown in Figure 5.3.

Should criterion (a) is applied, a maximum number of devices assigned per MA is also specified. The number of MAs required per PI is then automatically evaluated and their individual itineraries either manually or automatically specified. In the case that the number of devices is not a multiple of the number of MAs used, the MA object, which is launched last will be assigned a smaller set of devices. This design decision has been made to minimise the overall delay, as the MA object launched first has a marginal time advantage over the one launched last. Hence, should N network devices are divided in d management domains (d MA objects used in every polling interval), some MAs will be assigned to $\left\lceil \frac{N}{d} \right\rceil$ NEs and the remainder to $\left\lfloor \frac{N}{d} \right\rfloor$ NEs[1]. For instance, should 3 MAs are required for the management of 8 NEs, the first two will be assigned 3 NEs and the third only 2. Criterion (b) caters for the case that managed devices are distributed among several subnets, with criterion (c) being more suitable when a large number of NEs is concentrated in specific segments.

With the GnG approach, the launched MA objects travel and perform their management tasks independently, whilst the number of devices they visit is limited, thereby minimising the overall response time. This introduces a high degree of parallelism in the data collection process and suggests GnG as a suitable polling scheme for the acquisition of real-time data.

As the number of devices assigned per MA is reduced, the volume of MAs required to conduct polling is increased and the journey time of each of them decreased. Nevertheless, the manager needs more time and consumes more CPU cycles to instantiate, launch and receive back this increased number of MAs; a side-effect of launching and receiving a higher number of MAs is that the traffic in the manager’s network neighbourhood is also increased. The communication overhead is also affected, as short itineraries prevent MAs state from growing too much. However, for large number of domains, the number of MA transfers per PI increases proportionally, which in turn implies higher requirements on network resources. That represents an interesting trade-off, which signifies the need for a detailed investigation in order to identify optimal solutions.

[1] Precisely, $n = N - d\left\lfloor \frac{N}{d} \right\rfloor$ MAs are assigned to visit $\left\lceil \frac{N}{d} \right\rceil$ NEs and $d - n$ MAs to $\left\lfloor \frac{N}{d} \right\rfloor$ NEs.

5.2.1.1. Implementation of GnG Polling Scheme

GnG polling is carried out through *Polling Threads* (PT), implemented by the `Manager.PT` class. PTs are started and controlled by the manager application; each of them corresponds to a single monitoring task. They are created by the MAG tool, when defining the properties of the MA that carries out the task. In particular, the properties of a monitoring application are maintained as a *Polling Thread Configuration* (PTC) object (`Manager.PTC` class). In addition to keeping existing PTCs in memory, they are also serialised and stored in '*configuration files*'. In principle, the serialisation process is very similar to the one used by the MFC components during an MA migration; the two procedures differ only on that in the PTC case the serialised information stream is directed to a file rather than a network connection. Configuration files are updated every time that their corresponding PTC objects are updated and used at the manager application initialisation, in order to restore the PTCs state (through the inverse process of de-serialisation); that way, management tasks are defined only once.

PTs' functionality is determined by their respective PTCs, which comprise a description of the management application and include the polling parameters. Post their creation, PTs remain on 'waiting' mode until activated by the administrator. When activated, PTs define the polling task according to the requirements described in their respective PTC object and start the polling operation.

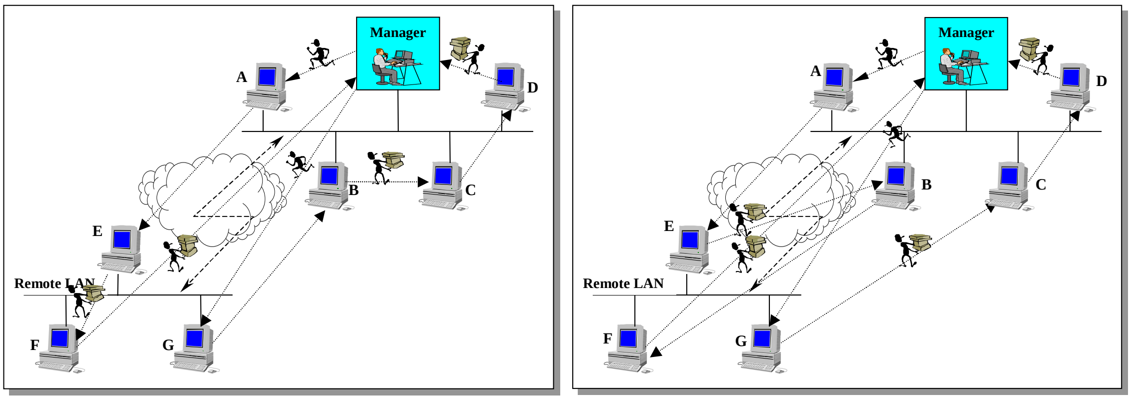
Specifically, when considering the GnG polling scheme, PTs instantiate and launch the required number of MAs (supplied with their corresponding itinerary) and then sleep for one PI; when that period elapses the same process is repeated. Meanwhile, a manager's listener daemon (Mobile Agent Listener thread) receives the MAs that return to the manager carrying their collected data. In a future extension, we will consider PTs synchronisation, so that monitoring tasks sharing the same polling frequency will not be initiated simultaneously. That will ensure that the traffic around the manager host is evenly distributed over time.

Polling properties may be modified at runtime. For instance, the PI may be adjusted, or new NEs may be added in the list of monitored devices. The discovery of a new active agent process triggers an automatic re-evaluation of the required number of MAs per PI and their itineraries; this procedure is transparent to the user.

A graphical table (within the manager's GUI) displays and allows modifications on existing PT properties, such as PT activation/de-activation, polling frequency and the number of devices assigned to each MA. That graphical component will be shown in Chapter 7 (Figure 7.13).

5.2.1.2. Optimal MAs Itinerary Planning

Despite the popularity of MA-based management applications, methodologies for designing efficient MA itineraries have received little attention. In particular, the number of hops realised by a multi-hop agent is not the only metric to evaluate the communication overhead of MA-based operations. The order in which MAs visit the NEs is also a crucial factor; slight changes on the agents' itineraries may result in dramatically variant management costs. The scenario illustrated in Figure 5.2 represents an ideal case in terms of the WAN link utilisation. That is because the link is traversed only twice per PI; the situation could be worse if partitioning criterion (a) was applied. For instance, network partitioning could be performed in such a way that the two MA objects would be required to visit NEs located in both the segment local to the manager station and the remote LAN. This is pictured in Figure 5.4a, with the first MA following the itinerary AEF and the second GBCD. In this case, the WAN link is traversed four times per PI. In a more pessimistic scenario, one of the MAs would not visit the remote LAN hosts in sequence, but traverse the link twice on each direction (itinerary: AEBF), as shown in Figure 5.4b. Apparently, itineraries scheduling lacks a mechanism that would guarantee minimal use of links connecting individual management domains, in other words *optimal itineraries planning* (OIP) is required.



(a) (b)

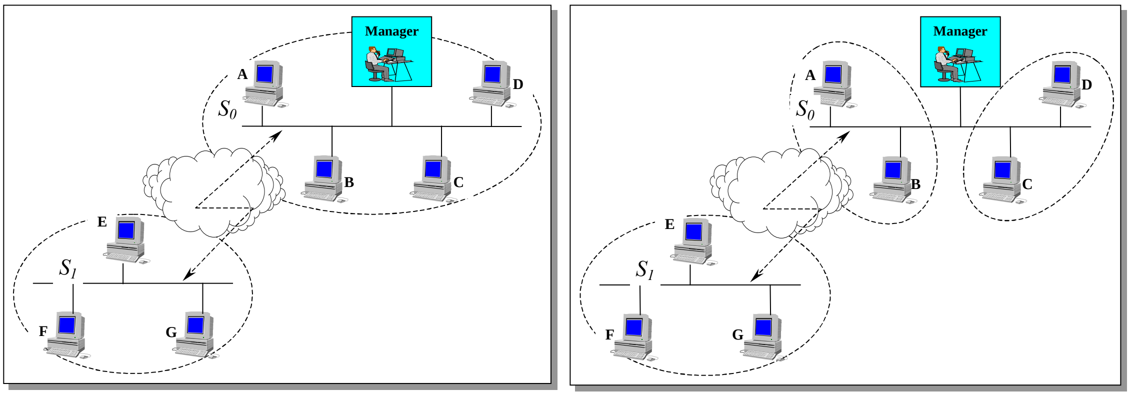
Figure 5.4. Non-optimised partitioning scenarios

Interestingly, the OIP problem exhibits many similarities with the Multi-point Line Topologies or Constrained Minimum Spanning Trees (CMST) problems. A CMST is a Minimum Spanning Tree (MST)[2] with the additional constraint on the size of the subtrees rooted on the centre. CMST algorithms are typically used in graph theory, with the main application area being network design problems. The most well known heuristics efficiently dealing with this problem are *Esau-Williams* (E-W) and *Sharma*'s algorithms [KER93]. These

[2] A MST is defined as a tree (i.e. a connected graph without cycles) with the least total distance, cost, or some other measure of delay or reliability.

algorithms propose near-optimal structures that minimise the total cost when a given network of *N* terminals is examined. Being heuristics, these algorithms cannot guarantee real optimal solutions; the latter would require the use of complex and time-demanding mathematical methods, such as Integer Linear Programming (ILP) [KER93]. However, for a relatively small number of terminals, the output of heuristics and ILP formulations typically coincide. The input of E-W and Sharma algorithms comprises the number *N* of terminals to be connected, the name of the central network site, constants $t_{i,j}$ and $c_{i,j}$ $(1 \le i, j \le N)$ that denote the traffic requirement and the cost for connecting terminals *i* and *j* respectively, and finally the aggregate cost C_{max} that sets a maximum limit for the cost of each individual multi-point line (subtree).

In principle, an algorithm dealing with the OIP problem would be a variation of a CMST algorithm. In particular, the managed network *V* would be a union of its *s* individual segments: $V = \{S_o, S_1, .., S_s\}$. The centre of the managed network (graph) would be the manager station (denoted as terminal *0*), located in segment S_0. The traffic constants would be: $t_{i.j} = 1, \forall i, j$, in order to eliminate their effect on the proposed solution. In addition, the cost constants would be $c_{i.j} = 0$ when $i \in S_k$ and $j \in S_k$ for any subnet S_k, while $c_{i.j} > 0$ when $i \in S_k$ and $j \notin S_k$. Clearly, in the latter case, the cost $c_{i.j}$ will not be a constant value but will reflect the cost of data transfers between two hosts located in separate network segments.



(a) (b)

Figure 5.5. Applying an OIP heuristic to propose (a) two, or (b) three near-optimal itineraries.

Using as case study the simple network topology illustrated in Figure 5.2, the employment of the algorithm briefly described above would prioritise the inclusion of the hosts located on segment S_1 in a single management domain, for both the cases that two or three optimal MA itineraries were requested (see Figure 5.5). In the extreme case that only one itinerary was requested, the application of the OIP algorithm would ensure that the hosts located in S_1 would be visited successively, namely the WAN link would only be traversed twice. Therefore, the algorithm described above would not only provide near-optimal clustering of the managed network in management domains, but also near-optimal solutions regarding the order in which individual MA objects should visit their assigned NEs.

Should the output of the algorithm not satisfies the requirements set by the administrator (regarding the desired number of NEs assigned to each MA), the C_{max} parameter could be adjusted (decreased), until a solution that incorporates the desired number of itineraries would be obtained. Alternatively, an additional parameter C_{min} could be used in conjunction with C_{max}.

The application of an OIP algorithm presupposes that the manager platform (which runs the algorithm) has detailed knowledge of its managed network topology, i.e. the physical location NEs. A simple managed network topology model is described in Chapter 6. It should be noted that the algorithm dealing with the OIP problem has not been implemented yet, but will certainly be considered among possible optimisations of our framework in future extensions.

5.2.2. Go ’n’ Stay Polling Scheme

GnS polling introduces an alternative approach to performance management, targeting data intended for off-line analysis. Within this approach, the mobility feature MA objects is not fully exploited, as the MAs intended to carry out decentralised management tasks are multicasted to the managed devices where they remain until their task is completed. This reduces the need for MA transfers, which are restricted on deploying MAs to remote devices. In addition, the proportion of *useful* management information returned to the manager is substantially increased, since NSM data are not necessarily delivered in every PI.

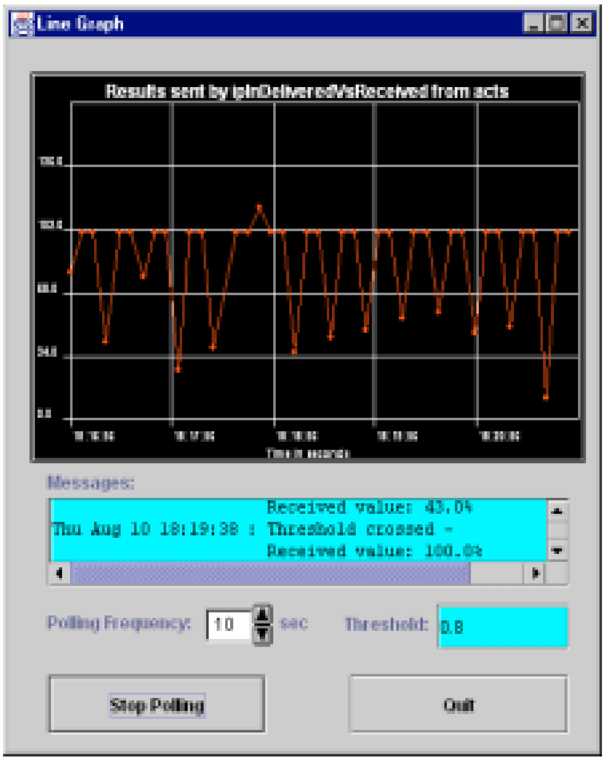
GnS approach may be useful for gathering performance information from a number of NEs, deliver performance reports on scheduled basis and event-driven notifications when performance thresholds are crossed. The user can analyse the performance reports and notifications obtained to determine utilisation trends, isolate performance problems, and possibly solve them before they cause non-reversible degradation of network performance. In this way performance monitoring can also aid in providing traffic flow predictions (per hour, day or month), long-term capacity and topology planning, identifying bottlenecks and congestion points, monitor quality of service (QoS) parameters, etc.

In principle, GnS polling scheme comprises a direct application of single-hop or constrained mobility (see Section 3.5.5.2). A prototype that implements a constrained mobility platform (CodeShell) has been introduced in [BOH00c] and described in Section 3.6.1.9.

5.2.2.1. Implementation of GnS Polling Scheme

The operation of GnS polling scheme is also controlled by PTs. Its implementation is much simpler than that of GnG approach, as the itinerary of multicasted MAs basically consists of only one host, while MAs are typically deployed only once during the monitoring task's lifetime.

Upon reaching its destination, an MA is instantiated and its execution subsequently started as a separate thread. Having the information of the PI duration embedded into its state, the MA obtains raw performance information from the local legacy system through interacting with the Service Facilitator (SF) component. The obtained data are processed, with the resulting higher-level information being encapsulated. The MA then ‘sleeps’ for a period defined by the PI duration (the `sleep()` method of the `java.lang.Thread` class is invoked). When this interval elapses, the MA ‘wakes up’, resumes its execution and obtains another data sample.



```
                      Received value: 43.0%
Thu Aug 10 18:19:38 : Threshold crossed -
                      Received value: 100.0%
```

Polling Frequency: 10 sec   Threshold: 0.8

Figure 5.6. Graphical representation of statistics returned by MAs employing GnS polling

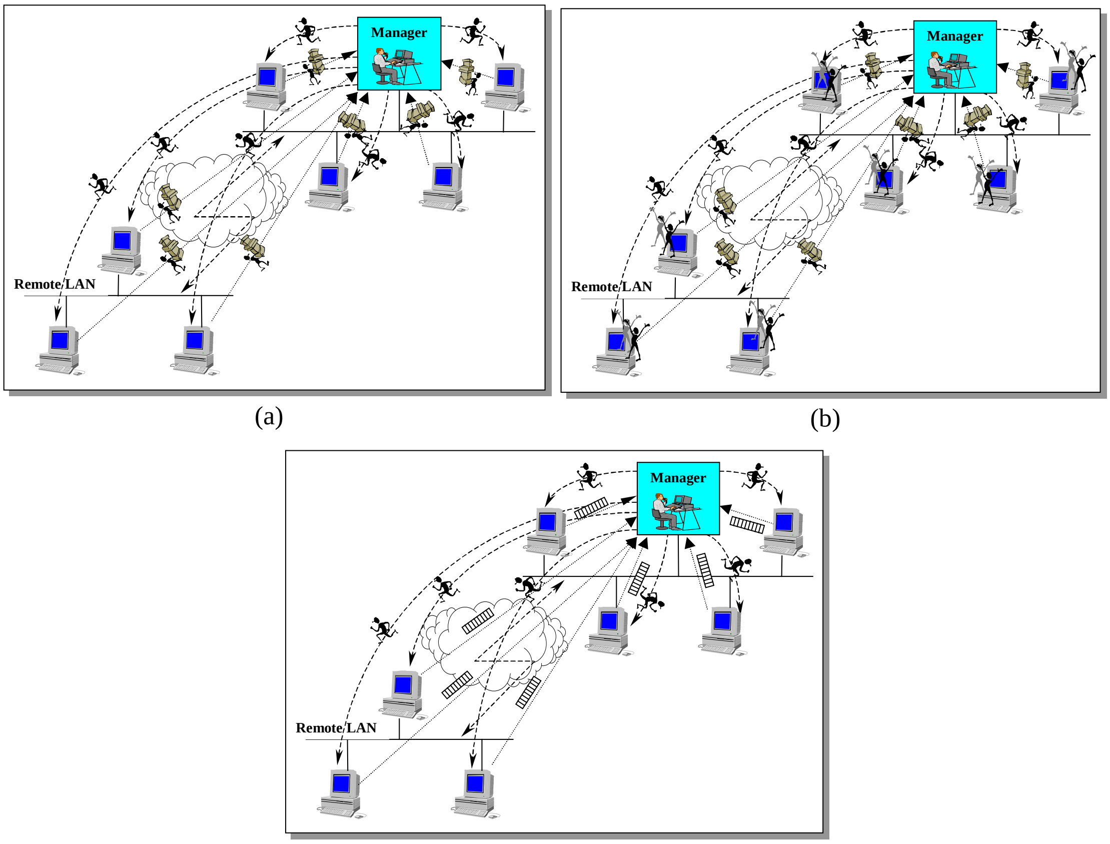
Collected samples are delivered to the manager station either in scheduled basis or in case that a monitored performance indicator exceeds a pre-determined threshold. In the latter case, a notification is instantly generated and sent to the manager platform. This functionality resembles the operation of metric objects [ISO93] and the summarisation function [ISO92], as specified in the corresponding standards. The data returned to the manager can be graphically displayed using graphs updated in real-time. This is illustrated in the graph shown in Figure 5.6, which depicts the fluctuation of a simple performance metric that combines two MIB objects over time (the metric value is evaluated at the managed devices by the MA objects that return the processed data samples).

MA objects either remain on their hosting NEs for a given number of PIs and then request the disposal of their execution thread, or execute in permanent basis (in the latter case the monitoring task will only be terminated at the event of a manager's request). Collected data may be delivered to the manager using three alternative approaches:

(a) through the multicasted MA objects themselves: PTs will repeat the deployment of the MAs to the NEs, given that the execution of the monitoring task is to be continued (see Figure 5.7a);

(b) through clones of the originally deployed MAs (see Figure 5.7b): MA clones are constructed through an explicit invocation of the `clone()` method of the `java.lang.Object` class;

(c) through an RMI call (see Figure 5.7c): the `returnData()` method of the `RMI.ManagerRmiServer` class is invoked.



(a)

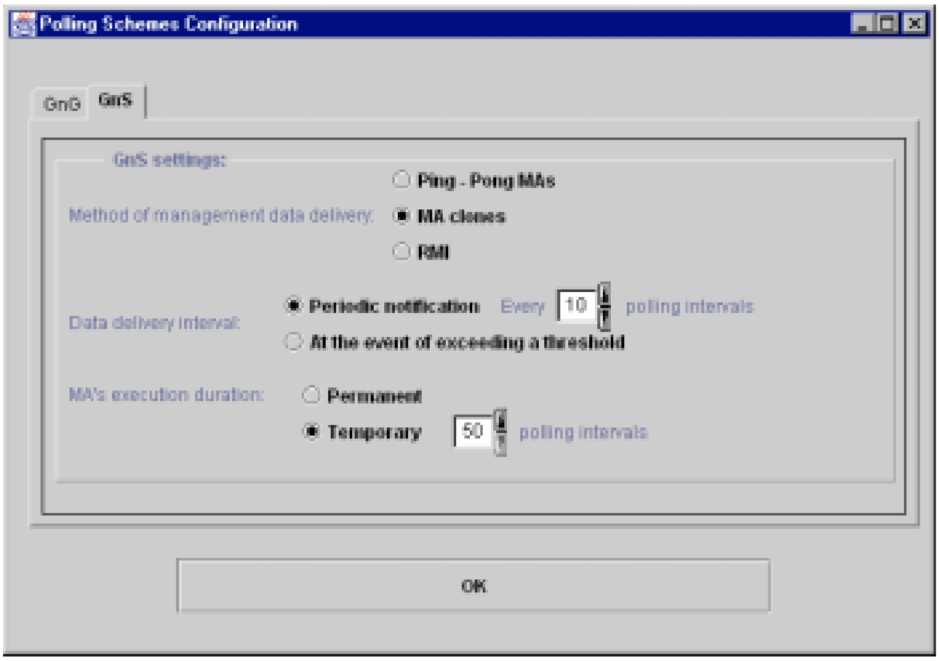
(b)

(c)

Figure 5.7. Approaches in GnS polling: data delivery through (a) the multicasted MA objects, (b) clones of the multicasted MAs, (c) RMI calls

When applying approach (a), the supervising PT multicasts at regular intervals an MA object to all monitored hosts. The MAs then remain active on the hosts for *p* PIs (where *p* is specified by the administrator). When the *p* PIs elapse, the MAs return to the manager to deliver the acquired samples. Meanwhile, PTs suspend execution for a period given by the product of PI and the number *p* of PIs that MAs remain on the managed devices ($PI \times p$). When this period expires, they resume operation and the process is repeated. In the extreme case that p=1, the GnS scheme performs similarly to SNMP-based polling and identically to GnG, when each MA is assigned to a single device. This data delivery method certainly implies a higher management cost, as it involves a larger number of MA migrations. However, it has been implemented for evaluation purposes. In approaches (b) and (c), the data returned to the manager either by MA clones or RMI invocations are discarded from the data folder of the MAs that remain on the NEs to continue their monitoring tasks; that way the usage of memory resources by locally executing MAs is moderated.

The properties of GnS polling are specified through the GUI shown in Figure 5.8.



GnS settings:

Method of management data delivery: ○ Ping - Pong MAs ● MA clones ○ RMI

Data delivery interval: ● Periodic notification Every 10 polling intervals ○ At the event of exceeding a threshold

MA's execution duration: ○ Permanent ● Temporary 50 polling intervals

Figure 5.8. Configuring the properties of GnS polling scheme

In the case of updating a monitoring task that employs the GnS scheme, the manager first requests the termination of executing MAs (through RMI calls to their hosting MAS servers) and subsequently deploys their updated version whose execution is immediately started.

It is noted that all the properties required to configure either GnG or GnS polling are declared in `MCode.Monitor` class, which extends the generic `MCode.MA` class. In other words, `Monitor` class defines the functionality of MA objects intended to perform decentralised monitoring tasks. Monitoring MA classes need to extend the `Monitor` class.

The administrator can modify the GnG/GnS parameters in real-time and also to change the polling scheme from GnS to GnG and vice versa, depending on the managed network traffic conditions and the types of management information required, without disrupting the monitoring procedure. The transition from GnS to GnG triggers an automatic network domain segregation process, which satisfies the chosen partitioning criteria.

5.3. Performance Analysis

In this section, simple mathematical formulations are devised in order to model the network overhead associated with the employment of the two proposed polling schemes. This performance evaluation essentially extends the evaluation of Section 4.5.2, using the same definitions, symbols and formalisms.

Considering the GnG scheme, we make the assumption that managed devices are equally distributed among individual MAs. Hence, the resulted overhead for GnG polling when segmenting the network into *d* domains would be,

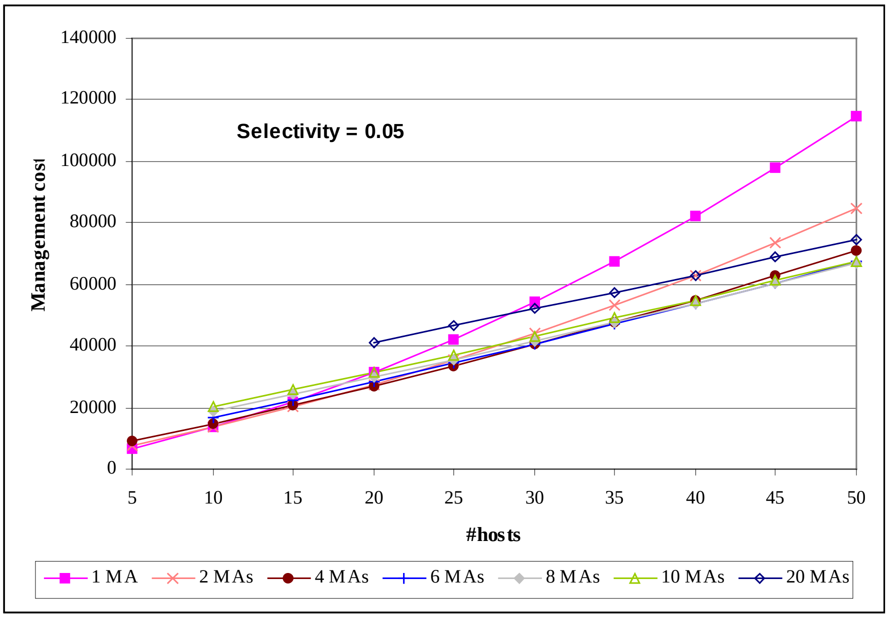
$$B_{GnG} = N*(C+O_T)+d*p*\sum_{h=1}^{h_{tot}}(ST_h+O_T), \text{ where } 1 \le d \le N \text{ and } h_{tot} = \left\lceil \frac{N}{d} \right\rceil + 1 \qquad (5\text{-}1)$$

where h_{tot} represents the number of hops for each MA object and the rest of the symbols hold their usual meaning. The first term of the equation describes the overhead imposed when multicasting the MA code to all MASs, whilst the second represents the bandwidth consumed by the MA state transfers between the manager and the polled devices (each MA is assigned to $\left\lceil \frac{N}{d} \right\rceil$ NEs, hence h_{tot} transitions in total, including the return to the manager). Substituting ST_h from Eqn. (4-7), we obtain:

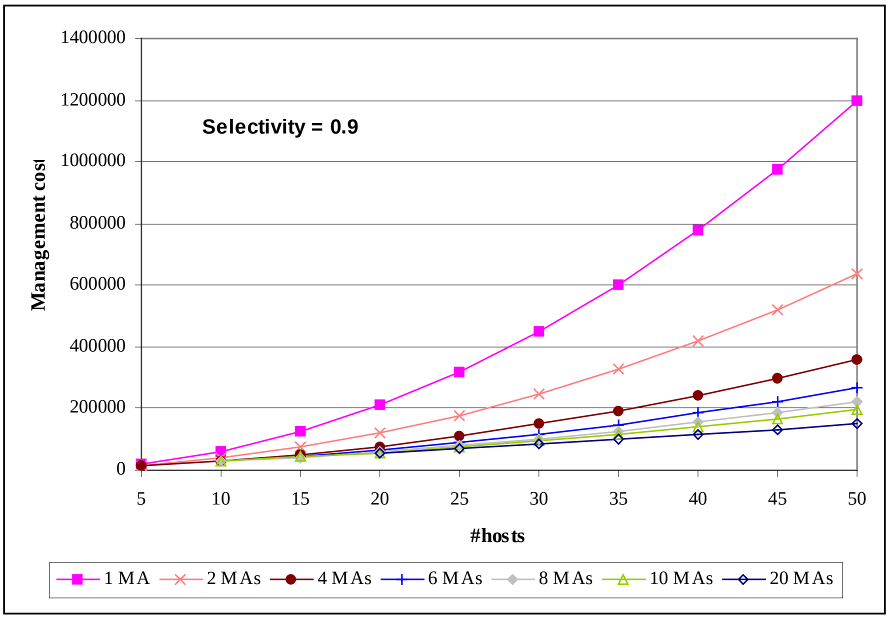
$$B_{GnG} = N*(C+O_T)+d*p*\left[(ST_0+O_T)*h_{tot}+\sum_{h=1}^{h_{tot}}(dS*h)\right], \ 1 \le d \le N \qquad (5\text{-}2)$$

where *dS* accounts for the state size increment experienced on every MA hop ($dS = \sigma * b$). The conclusions inferred for multi-hop MA-based polling are valid for GnS polling also, although a larger number of MAs (equal to *d*) are transferred on each PI. In particular, the GnG scheme involves $d*h_{tot} \approx N+d$ MA migrations, compared to *N*+1 involved in flat MA-based polling. However, the former is expected to perform better in case that the managed network is fairly large, selectivity values are high or large amounts of data are collected from each host. In that case, the network segmentation will result in shorter itineraries, enforcing travelling MAs to return back to the manager station before their state becomes prohibitively large.

This is graphically illustrated in Figure 5.9, which depicts the dependence of management cost to network size and scalability values. As in Section 4.5.2, we assume an initial MA state size of 400 bytes, 1000 bytes of retrieved data and that *no* data compression is performed. Therefore, for low selectivity values (σ = 0.05, i.e. only the 5% of the originally obtained information is returned to the manager), using a large number of co-operating MAs (e.g. 20) does not represent an appropriate solution, at least for small and average-sized networks (see Figure 5.9a).



(a)



(b)

Figure 5.9. Management cost for GnG polling scheme, as a function of the network size and the number of management domains (i.e. number of MAs used per PI), for selectivity equal to (a) 0.1, or (b) 0.9

The distinction is clear in the case of high selectivity values (see Figure 5.9b) where the separating gap between flat MA-based management (using one MA) and GnG polling is magnified. It is also evident that there is an optimal number of MAs associated with a given network size, which minimises the management cost. For σ = 0.9, that number is 3 MAs for a network of 5 managed hosts, 6 MAs for 10 hosts, 8 MAs for 15 hosts, 10 MAs for 20 hosts, etc. In our current prototype, the decision regarding the number of management domains is made by the administrator, however, in a future extension we intend to provide automated adaptation of that number to the size of the managed network in order to minimise the associated network overhead.

Similarly to GnG, the network overhead for GnS polling scheme would be:

$$B_{GnS} = N * (C + O_T) + N * \left(ST_o + O_T\right) + B_{del} \tag{5-3}$$

where the first term of the summation represents the bytecode distribution, the second the actual MAs deployment and the third the bandwidth wasted for data delivery. Only the last term exhibits dependence on the time interval over which the monitoring task is executed. Assuming that data are periodically delivered every p' PIs through the same MA objects that were originally multicasted, B_{del} is given by:

$$B_{del} = N * \left[2 * (ST_0 + O_T) + dS'\right] * \left\lceil \frac{p}{p'} \right\rceil \tag{5-4}$$

The MAs return p' data samples, which result in a state size increment of dS'. Following the delivery of management information, a new set of MAs will be multicasted again (hence the *2* multiplier). For a large p', the number of MA transfers is minimised and GnS mode becomes more lightweight in terms of bandwidth consumption. In the case of delivering data through clones of the original MAs, B_{del} becomes:

$$B_{del} = N * \left(ST_0 + dS' + O_T\right) * \left\lceil \frac{p}{p'} \right\rceil \tag{5-5}$$

Namely, the deployment of a new set of MAs every time that data are delivered is not further required. Accordingly, data delivery through RMI is modelled as follows:

$$B_{del} = N * Rmi(dS') * \left\lceil \frac{p}{p'} \right\rceil \tag{5-6}$$

where $Rmi(dS')$ denotes the traffic generated by an RMI call that passes as an argument a data vector of size dS'.

A quantitative evaluation of response time for MA-based operations is not attempted here as the adoption of GnG introduces additional complexity which makes its mathematical modelling very hard, whereas modelling response time for GnS polling would be meaningless since reducing latency is not a prime objective for this scheme.

5.4. EXPERIMENTAL RESULTS

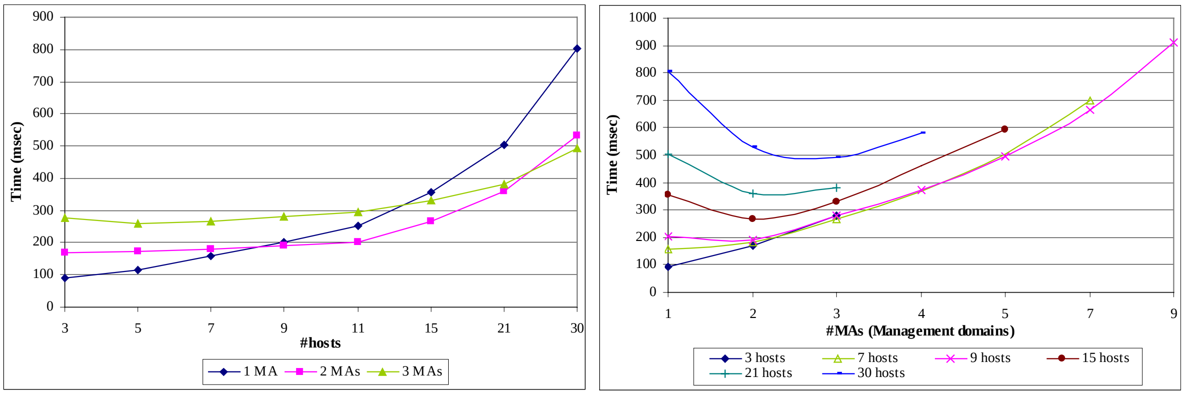
In this section, we describe a number of experiments aiming at evaluating the performance of the introduced polling schemes and comparing it against flat MA-based polling. The physical distribution of managed devices is assumed as arbitrary to these experiments. Similarly to the previous chapter, latency and network overhead measurements are presented.

5.4.1. Response Time Measurements

One of the main motivations that led to the design of GnG polling scheme has been to reduce the response time of flat MA-based polling implementations. Hence, the investigation of the factors affecting latency is of great importance. These factors include the network size, the amount of collected data and the number of MAs employed in each PI. TCP is used for MA transfers, while no data compression is applied. Practically, we have repeated the experiment of Section 4.6.1.2, which investigates the performance of multi-hop MAs, i.e. flat MA-based polling. Each travelling MA sequentially visits the polled devices included into its management domain, obtaining a certain amount of information at each point of contact, which can either be 50 (see Figure 5.10) or 2000 bytes (see Figure 5.11), before returning to the manager station to deliver their collected data.

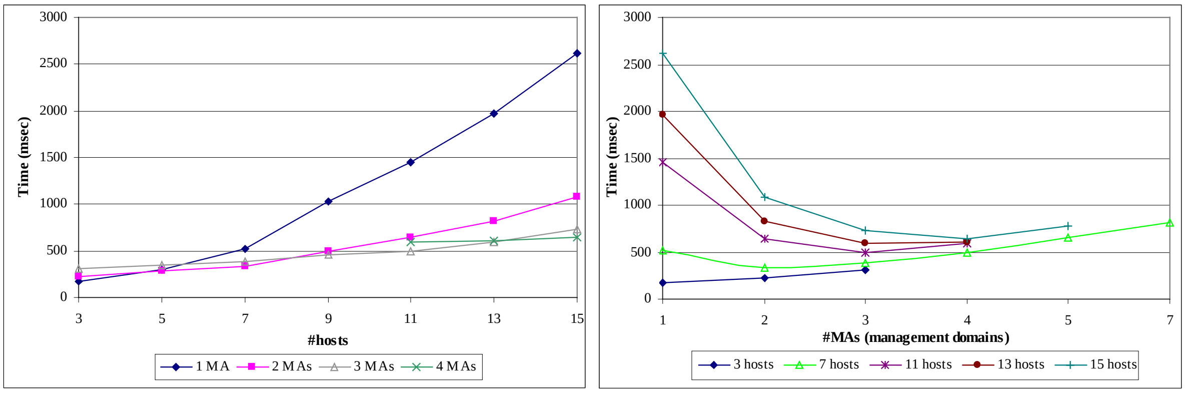
Figure 5.10a shows that the flat approach (using a single MA) does not scale well as the number of NEs increases. This makes it necessary to partition the managed network into several domains in order to maintain low overall response time. In particular, it is shown that in the case that only 50 bytes are collected from each host, using 2 MA objects improves efficiency for networks including more than 13 devices. Likewise, 3 co-operating MAs perform better for networks of 24 managed devices or more. Interestingly, the respective thresholds differ in the case that the amount of collected data equals 2000 bytes. In particular, the transition to two or three co-operating MAs models would improve performance for a number of hosts bigger than 4 and 9 respectively (see Figure 5.11a). This is due to the faster growth of MAs state size, which in turn affects their transfer latency.

Depending on the managed network size, the optimum number of domains (number of MAs working in parallel) can be determined from the minimum point of the corresponding curves of Figure 5.10b and Figure 5.11b. All curves displayed in these figures are convex, suggesting that the optimal solution always lies between the extremes of segmenting the network in very few or too many management domains. Namely, it is shown that assigning a small number of devices to each MA, i.e. launching a large number of MAs per PI typically increases the overall response time. That is, although the individual journey times decrease, the time required for the manager to instantiate, launch and receive back these MAs dominates. Thus, for a network of 15 devices, when collecting 50 bytes from each host, response time is minimised when partitioning the network in 2 domains, i.e. assigning 8 NEs to the first MA and 7 to the second. Ideally, the manager application (through the PTs) should dynamically adjust the number of domains according to the current number of managed devices, aiming at maximising efficiency. The conclusions reported in this section agree with the simulation results presented in [RUB00], which investigates strategies for reducing the delay involved in MA-based performance management applications.



(a) (b)

Figure 5.10. Polling response time of GnG polling in the case that 50 bytes are collected from each host, as a function of (a) the network size, (b) the number MAs launched per PI



(a) (b)

Figure 5.11. Polling response time of GnG polling in the case that 2000 bytes are collected from each host, as a function of (a) the network size, (b) the number MAs launched per PI

The information graphically presented in Figure 5.10 and Figure 5.11 is analytically presented in Table 5.1 and Table 5.2 respectively.

# MAs launched per Polling Interval

	1		2		3		4		5		7		9	
# Polled Devices	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
3	91.4	5.9	169.8	11.6	278.7	18.9	-	-	-	-	-	-	-	-
5	114.6	5.4	173.4	19.7	260.4	15.6	-	-	487.8	41.3	-	-	-	-
7	158.8	5.7	180.9	22.7	265.4	19.7	-	-	-	-	698.3	77.8	-	-
9	202.9	13.1	189.1	23.6	281.0	26.2	-	-	-	-	-	-	912.0	81.3
11	250.7	5.4	202.1	19.1	294.3	40.6	-	-	-	-	-	-	-	-
15	355.9	25.4	268.2	22.0	331.3	38.2	-	-	613.5	45.2	-	-	-	-
21	504.5	30.3	358.4	45.1	382.7	45.7	-	-	622.9	47.1	-	-	-	-
30	803.5	64.1	531.4	81.3	493.5	83.4	579.6	124.8	-	-	-	-	-	-

Table 5.1. Average polling time and standard deviation for GnG polling, for multi-hop MAs for various network sizes and numbers of management domains (50 bytes are collected from each host).

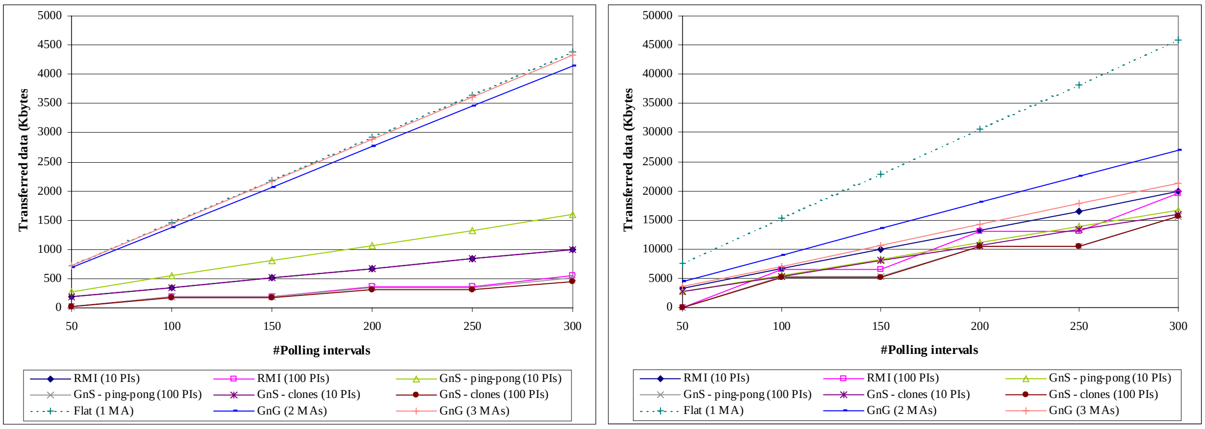
# MAs launched per Polling Interval

	1		2		3		4		5		7		9	
# Polled Devices	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
3	176.7	35.7	220.7	14.0	309.9	24.6	-	-	-	-	-	-	-	-
5	300.4	36.3	280.5	37.2	344.1	42.0	-	-	549.4	46.7	-	-	-	-
7	521.0	49.5	335.0	37.3	381.2	29.2	-	-	-	-	816.0	86.7	-	-
9	1030.3	46.8	495.9	26.5	454.8	33.3	-	-	620.2	-	-	-	1129.5	143.6
11	1452.3	36.3	638.5	28.6	496.7	48.1	596.9	49.9	-	-	-	-	-	-
13	1967.9	68.0	821.1	37.4	590.6	61.9	605.9	58.9	-	-	-	-	-	-
15	2611.5	71.8	1084.2	54.5	728.9	80.9	647.5	74.6	779.3	48.0	-	-	-	-

Table 5.2. Average polling time and standard deviation for GnG polling, for multi-hop MAs for various network sizes and numbers of management domains (2000 bytes are collected from each host).

5.4.2. Network Overhead Measurements

This section reports the results of a simple experiment that aims at measuring the network traffic incurred when the management of a small set of devices is involved. The experimental testbed comprises a PC that plays the role of the manager and 10 PCs in which a MAS server is installed, simulating managed devices. We compare the performance of GnG polling (for various network partitioning configurations) against that of GnS (for various data delivery frequencies).



(a) (b)

Figure 5.12. Network traffic incurred by GnG, GnS and RMI approaches as a function of the polling intervals for data samples equal to (a) 50 bytes, and (b) 2000 bytes.

# Polling Intervals	RMI (10 PIs)	RMI (100 PIs)	GnS (ping-pong, 10 PIs)	GnS (ping-pong, 100 PIs)	GnS (clones, 10 PIs)	GnS (clones, 100 PIs)	GnG (1 MA)	GnG (2 MAs)	GnG (3 MAs)
	Transmitted data on MAC layer, Kbytes (data sample: 50 bytes)								
50	183	20	281	20	183	20	729	689	722
100	346	194	543	184	347	165	1459	1379	1443
150	509	194	805	184	511	165	2188	2068	2165
200	672	369	1067	349	675	309	2917	2758	2887
250	836	369	1329	349	838	309	3647	3447	3608
300	999	544	1591	513	1002	454	4376	4137	4330
	Transmitted data on MAC layer, Kbytes (data sample: 2000 bytes)								
50	3326	20	2788	20	2690	20	7640	4502	3558
100	6632	6518	5557	5243	5361	5224	15281	9004	7117
150	9939	6518	8326	5243	8031	5224	22921	13507	10675
200	13245	13017	11094	10467	10702	10428	30562	18009	14233
250	16551	13017	13863	10467	13373	10428	38202	22511	17792
300	19858	19515	16632	15691	16043	15632	45843	27013	21350

Table 5.3. Comparison of GnG, GnS and RMI-based approaches in terms of network overhead: The volume of the transferred data on the MAC layer

In particular, network traffic is measured (by the Windump tool) for the cases that single data samples comprise 50 or 2000 bytes. The performance of GnG polling is investigated when employing 1 (flat management), 2 or 3 MAs. When considering GnS polling, data delivery frequency can either be 10 or 100 PIs, i.e. 10 or 100 data samples respectively are returned to the manager either through an RMI call or encapsulated into an MA state. Focusing to GnS polling, data deliveries are performed either by the originally multicasted MAs, their clones or RMI calls. The results graphically illustrated in Figure 5.12 are also analytically presented in Table 5.3.

As expected, the presented results demonstrate a clear advantage of GnS over GnG polling scheme, especially when the monitoring tasks execute for long time periods. The separating gap is, however, reduced for large data samples (see Figure 5.12b). In both cases, flat MA-based polling exhibits the worse performance amongst all alternative approaches. Regarding

GnS polling, data deliveries through clones of the originally deployed MAs has been shown to represent the most efficient approach. This conclusion agrees with the results presented in Section 4.6.2, according to which our MA framework marginally outperforms RMI in terms of network overhead when considering simple data transfers between a pair of hosts. Furthermore, extra savings on GnS overhead may be achieved by applying more infrequent data deliveries (every 100 instead of 10 PIs).

5.5. DISCUSSION

It should be already evident that the decision regarding the choice of the appropriate polling scheme depends on a number of factors, such as the type of the monitoring task, response time restrictions, the estimated task execution period, the physical distribution of managed devices, etc. The following sections comprise an analytical comparison of the two polling schemes, discussing the advantages and trade-offs associated with their employment.

5.5.1. GnG polling scheme

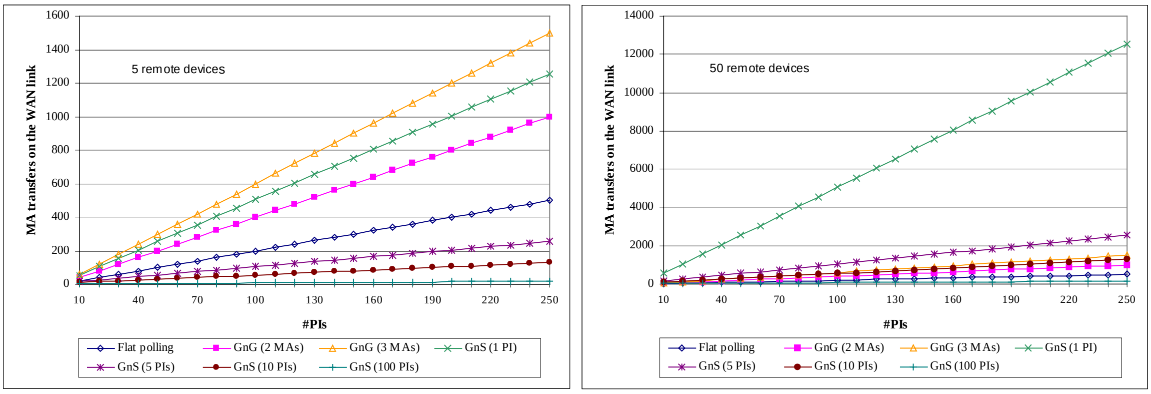
GnG polling represents an efficient management model in the case that real-time operations are involved, namely when response time restrictions apply. Specifically, it has been shown that for relatively large networks, applying logical network partitioning and letting a set of MAs visiting their assigned nodes in parallel, may significantly improve efficiency in comparison with both flat MA-based model (launching a single MA) or deploying an MA to every device. In the latter case, each MA would typically be executing for a time negligible with respect to its deployment time; hence, the agent deployment overheads would be unacceptable for time sensitive applications [BOH00b].

GnG polling also results in network overhead savings compared to the flat model, especially when the amount of data stored within MAs state is substantial. When the monitoring task is intended to run for a short time period, GnG polling may achieve network traffic savings even over GnS mode.

GnG polling also relaxes the management station from processing bottlenecks that would appear should GnS polling be applied for the management of a large number of NEs (in such case, an equal number of MAs would be instantiated and launched, resulting in overloading the manager platform). An invited side-effect of using a relatively small number of MAs is that network capacity around the manager platform is not saturated by the simultaneous generation and transmission of the agents, leading to more even utilisation of network resources. It should be acknowledged though that flat management answers the two aforementioned problems (manager overloading and management traffic distribution) more adequately, as only one MA object is transmitted in every PI.

Furthermore, modifications of monitoring tasks are easier when employing GnG polling. That is because the corresponding MA classes are updated on the manager station, with the modifications taking effect on the PI following the update. On the other hand, when using the GnS approach, MA updates are carried out through deploying the new version of the modified MA classes; frequent modifications may result in a dramatic increment of network overhead.

Last, GnG polling may also reduce the utilisation of links connecting individual network segments. If, for instance, a topology similar to the one illustrated in Figure 5.2 is considered, the employment of GnG polling would allow the deployment of a single MA object sequentially visiting all the NEs located within the remote subnet, before returning to the manager station. If, on the other hand, GnS polling is employed, both MA deployments and data deliveries will make use of the WAN link. Figure 5.13 depicts the number of MA transfers over the WAN link for various GnG and GnS configurations. Clearly, when a large number of NEs is concentrated in the remote subnet, GnS polling should apply infrequent data deliveries to maintain low link utilisation (see Figure 5.13b).



(a) (b)

Figure 5.13. Number of MA transfers over the interconnecting link required for the management of a remote subnet including (a) 5, or (b) 50 devices

5.5.2. GnS polling scheme

GnS polling can be thought of as a natural evolution of the REV paradigm where the uploaded MA code does not only represent a remote service, but can also act as a fully autonomous software component. It is therefore particularly suited to programming and dynamically extending the capabilities of network devices.

With respect to monitoring applications, GnS polling represents an attractive distributed management model for off-line analysis of bulk management data. In particular, its use is advantageous over GnG approach in cases that monitoring tasks are intended to run for relatively long time frames. Since it performs MAs deployment only once, it practically diminishes MA transfers through the network, reducing the associated traffic, given that data deliveries are not very frequent. For certain classes of monitoring applications, when the manager should be notified only on emergency situations, GnS scheme may reduce the bandwidth usage even further as the manager is notified only when certain thresholds, referring to performance metrics, are exceeded.

Furthermore, since management operations are performed locally, the polling frequency of monitoring tasks can be increased (i.e. the PI interval decreased) without putting any additional strain on network resources. That would allow observing instantaneous fluctuations of network & systems performance, providing more accurate statistics as the values of performance indicators would not be averaged over long time periods. Increments of polling frequency should however be regulated so as not to result in excessive consumption of system computational resources.

5.6. SUMMARY

This chapter introduced two polling schemes that aim at answering the scalability problems of flat MA-based management. Their implementation is based on the MAP presented in Chapter 4. The two polling schemes complement each other, in the sense that they jointly cover a wide range of monitoring tasks.

In particular, GnG polling is suitable for collecting on-line data and performing simple control and configuration tasks on several NEs. Response time is minimised by employing a number of MAs that travel within their assigned management domain, introducing a high degree of parallelism in the data collection process. This method may also be preferable when considering the management of remote LANs for short time range as it can reduce MA transfers over the interconnecting links. Concerning the off-line analysis of management data, we propose the GnS polling scheme. In this approach, MAs collect a larger amount of data before delivering performance reports to the manager, leading to drastic reduction of MA transfers. Hence, the selection of the appropriate polling scheme (and its associated parameters), is a compromise between network overhead, response time, execution period, etc, depending primarily on the type of management data to be collected. In both cases, semantic compression of NSM data can be applied to further reduce bandwidth usage. The performance analysis and results indicate a significant improvement in both response time and traffic overhead when comparing the introduced polling schemes to flat MA-based polling.

As a final remark, it should be emphasised that despite the improved scalability that GnG and GnS polling schemes introduce over centralised and flat MA-based management, the problem of managing remote subnets is not adequately addressed. Specifically, when physically and geographically dispersed managed network topologies are involved, the application of the two polling schemes implies a rather heavy utilisation of the interconnecting links. The management system has therefore increased dependency on network resources, whilst being prone to network failures. Furthermore, the manager station represents a single point of failure, as it is still responsible for managing the entire managed network, regardless of its size and topology structure. All these issues suggest that there is still space to further improve MA-based management scalability and flexibility. Concrete ideas for achieving such improvements are proposed in Chapter 6.

CHAPTER 6

ADAPTIVE HIERARCHICAL MANAGEMENT

6.1. INTRODUCTION

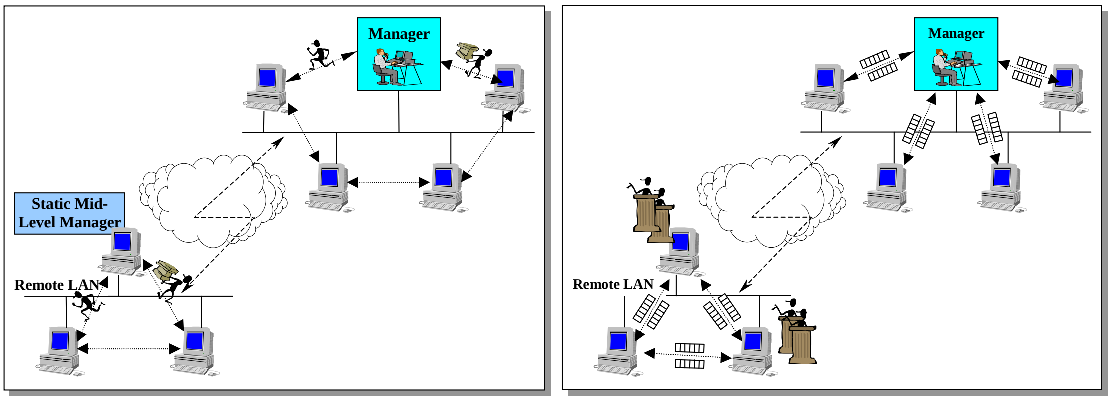
Contemporary large enterprise networks follow a hierarchical structure, spanning applications, organisational and geographical boundaries. In order to cope sufficiently with the unpredictable growth of the number of network devices, logical partitioning is being employed as a design and deployment principle. Any of the following or a combination of these partitioning criteria can be used: (a) physical distribution, i.e. partitioning based on the 'geographical' location of network elements (NE); (b) administrative subdivisions; (c) grouping based on different access privileges and security policies; (d) performance-driven network partitioning, e.g. aiming at evenly distributing traffic load among network segments. The design of such networks' management systems cannot escape that rule: the resultant network clusters may be better managed by hierarchically structured management models.

However, there is a notable inconsistency between the topology of hierarchically structured networks and the organisation models of emerging Mobile Agent (MA)-based management frameworks, the majority of which insists on 'flat' architectures. That approach fails to solve the scalability limitations of centralised architectures as it has been shown to result in increased response time and network overhead (see Section 4.5). The situation seriously deteriorates when considering management of remote LANs, connected to the backbone network through low-bandwidth, expensive WAN links. In this case, frequent MA transfers are likely to create bottlenecks and affect the overall management cost.

The response time part of flat management scalability problem is addressed by the 'segmentation' approach, i.e. Get 'n' Go (GnG) polling, introduced in Section 5.2.1; this scheme enables the parallel and therefore more time efficient data collection. The network overhead problem is more efficiently addressed by the 'broadcast' approach, i.e. Go 'n' Stay

(GnS) polling, described in Section 5.2.2; this approach is not however suitable for real-time operations or monitoring tasks whose execution is planned for short-time periods.

Geographically dispersed enterprise networks are more efficiently managed by hierarchical management systems (presented in Sections 2.10 and 2.11) which obviate the need for centralised Simple Network Management Protocol (SNMP) polling and localise management traffic through mid-level manager (MLM) entities acting on behalf of the central manager. However, existing hierarchical management solutions imply an inflexible and static definition regarding the physical location where MLMs execute and the assignment of the managed devices under their supervision (see Section 2.12). In other words, they are only suitable for static network topologies but are not in step with the dynamically evolving topological and traffic characteristics of large-scale enterprise networks. MA technology could provide the means for enriching the functionality and improving the flexibility of hierarchical management [MOU98a].



(a) (b)

Figure 6.1. Approaches to hierarchical MA-based management

The concept of hierarchical MA-based management is not entirely new. In particular, two approaches have been presented so far: (a) Use of static MLMs relying on MAs for the network monitoring process [LIO98, SAH98] (see Figure 6.1a); (b) Use of mobile MLMs performing decentralised SNMP management [ZAP99] (see Figure 6.1b). The first approach does not adequately address the flexibility limitations of proprietary hierarchical management. Still, even the latter approach lacks mechanisms for achieving automatic adaptation of the management system to changing network configurations, i.e. MLMs do not normally change the location where they execute. This model also involves the deployment of a new MA for each introduced monitoring task; these MAs work independently and do not necessarily execute at the same host. However, this approach is not in line with the concept of a *compact* MLM entity responsible for *all* the decentralised operations performed within its domain, which in our opinion offers better grouping, organisation, manageability and control over distributed Network & Systems Management (NSM) tasks.

The aforementioned approaches have certainly realised a step forward, yet, their direct application to distributed management is not straightforward. Critical issues such as well-defined criteria for segmenting the network into management domains, explicit determination of the domain boundaries or strategies for assigning mid-level managers to these domains, are not elucidated. Although the problems that need to be solved have been identified, the rules that define the mid-level entities' deployment strategy, i.e. questions concerning *when* and *where* to deploy, remove or change the location of mobile mid-level managers still remain open. Furthermore, it is important to use MAs both as MLMs and for the actual monitoring process in order to take advantage of their ability to filter management information at the source. The development of a highly adaptive hierarchically structured MA-based management model seems a rational approach to address these issues as well as to overcome the limitations of statically configured NSM frameworks.

This chapter addresses these issues through introducing a highly scalable and adaptive hierarchical management model. Such a model presupposes the presence of a novel management element, termed the *Mobile Distributed Manager* (MDM), operating at an intermediary level between the manager and the stationary agents. MDMs are essentially MAs, which take full control of managing a specific network domain and localise the associated management traffic. Apart from the fact that management functionality may be added/configured at runtime, this architecture can also dynamically adapt to fluctuating networking conditions. The system's scalability is further improved by assigning monitoring tasks to additional MAs (launched and controlled by the MDM) capable of filtering collected data locally. MDMs are deployed to remote subnets according to policies defined by the administrator.

The ideas introduced in this chapter have been originally published in the proceedings of the *7th International Conference on Intelligence in Services and Networks* (IS&N'2000), with further extensions and implementation details published in the proceedings of the *IEEE Global Communications Conference* (Globecom'2000). A brief overview is also included in a paper to appear as an invited contribution in the *Microprocessors and Microsystems* special issue on "Mobile Agent Technology: from first proposals to current evolutions". An extended version of the two conference papers has been submitted to *Computer Networks* journal. Full references are given in Appendix A.

The remainder of the chapter is organised as follows: Section 6.2 describes and explains the rationale behind our chosen hierarchical approach. Section 6.3 comprises the core of the chapter, discussing the implementation details of the introduced architecture. The advantage of the proposed model over non-hierarchical models is verified by a quantitative evaluation of the bandwidth usage, given in Section 6.4. The evaluation is complemented by experimental results reported in Section 6.5. Finally, the chapter is summarised in Section 6.6.

6.2. Hierarchical, Mobile Agent-based Network Management

In search of more flexible solutions, this work aspires to push the concept of MA-based hierarchical/distributed management much further. Specifically, we introduce the novel concept of MDM, referring to a management component that operates at a level between the manager and management agent end points. MDM entities are mobile objects that undertake the full responsibility of managing a network domain, when certain criteria (determined by the administrator) are satisfied. Upon being assigned to a domain, the MDM migrates to a host running in that domain (Figure 6.2a) and takes over the management of local NEs from the central manager.

As a result, the traffic related to the management of that domain becomes localised, as the MDM is able to dispatch and receive MAs to collect NSM data from the local hosts (Figure 6.2b), or even execute centralised management operations upon them. The MDM continues to perform its tasks without the manager's intervention, even if the interconnecting link fails. A first-line response can also be given to tackle trivial faults/alarms, with the manager being notified only in case of a complex problem or emergency situation. In performance management applications, only aggregated values and statistics are sent to the manager at regular intervals, thereby diminishing the amount of data transferred through the WAN link. The duration of these intervals is application-dependent and determined by the administrator.

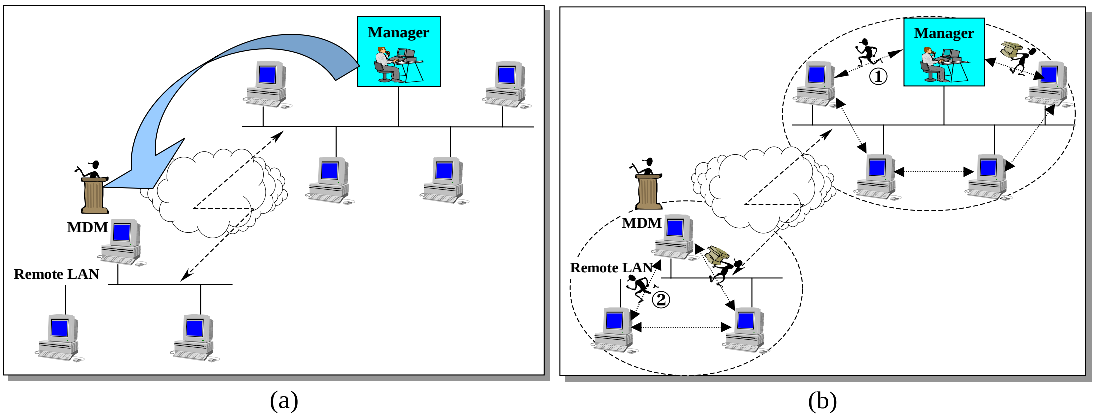


Figure 6.2. Hierarchical MA-based management

The decision concerning the selection of the host where the MDM will carry out its management tasks from, will be discussed in Section 6.3.6. It is noted that the management domain assigned to an MDM entity may be confined to a single network segment or expand to a larger set of hosts.

The mobility feature of MDMs allows the network management system (NMS) adaptation to network dynamics, optimising the use of network resources. Management functionality can be downloaded at runtime, while this architecture can also dynamically adapt to changing networking conditions. Namely, an MDM entity can be deployed to / removed from a network segment in response to a change in network traffic distribution, or move to the least loaded host to minimise the usage of local resources.

Notably, should the management domain assigned to an MDM comprises a large set of nodes, scalability problems might arise if centralised or flat MA-based management are employed for the data collection process. It is therefore important to further improve NSM scalability by combining our hierarchical framework with GnG/GnS polling schemes. For instance, MDMs can be easily coupled with GnG scheme: after deploying the MDMs to their remote domains, these can launch sufficient number of MAs per polling interval (PI) to reduce the overall response time. Alternatively, coupling the hierarchical model with GnS polling would minimise MA transfers within individual domains, thereby reducing the localised traffic volume.

The adaptive hierarchical management model is particularly suited to dynamically evolving networking environments. Namely, when the administrators perform frequent changes on network configuration, e.g. transfer network devices from one site to another or divide network segments in two or more subdivisions by installing interconnecting devices (bridges, routers, etc). The adaptability feature is also useful in cases that the manager platform is not permanently connected to a specific network location. To illustrate, let us consider an enterprise network comprising two separate segments, managed by a single manager application, possibly running on a portable computer. Should the administrator disconnect that computer from its current location and connect it to another site, the manager application might consider (depending on the chosen management policies) re-organising the NMS, i.e. re-defining the domain boundaries, terminating remote MDMs and deploying new ones to different locations, etc. In the mean time, the management operation will not be affected.

Summarising, the proposed architecture meets the following design requirements:

- *Load balancing*: The total workload should be equally distributed among the various processors of the underlying subsystems. MAs can take full advantage of the increasing processing capability of network devices to achieve management intelligence distribution, however that should not lead to exhaustive consumption of local resources. Therefore, MDMs should be designed as lightweight as possible so as to have minimal footprint on their hosting devices, i.e. they should be equipped with basic management functionality.

- *Fault-tolerance and robustness*: MDM entities should be programmed in such a way so that when detecting a failure on the inter-connecting link between their local subnet and the manager's site or a failure on the manager platform itself, to continue performing their decentralised tasks as normal. As soon as the communication channel is restored, all management data collected in the meanwhile can be returned to the manager.
- *Minimal intrusiveness*: MDMs should be deployed at specific hosts so as to minimise their intrusiveness in terms of the effect of management-related traffic on other applications and the additional processing burden placed upon host processors.
- *Dynamic adaptation*: Topological and traffic characteristics of modern networks are rapidly changing. A hierarchical NSM system should therefore be flexible enough to adapt to those changes. Hence, the location where MDMs run is not fixed, neither is the set of hosts under their control. MDMs can be transparently sent to a domain when the associated cost savings are considerable or removed when their existence is no longer necessary. They can also autonomously decide to move within their domain when the host processor is overloaded and continue their operation on the least loaded node.

6.3. IMPLEMENTATION DETAILS

The hierarchical model introduced herein has been implemented as an extension of the core MAP presented in Chapter 4. The following sections discuss in detail the implementation aspects of the model.

6.3.1. Topology Tree and Topology Map of Active Devices

An important element of the hierarchical framework is the *topology tree* (implemented by the `Manager.TopologyTree` class), a tree structure that comprises a representation of the managed network, made available to the manager application. In particular, the topology tree represents the underlying network topology, namely the individual subnets, the devices physically connected to these subnets and the way the latter are interconnected. It also provides information about the devices hosting active Mobile Agent Servers (MAS). Each of the topology tree nodes corresponds to a specific subnet, with the node representing the manager platform’s location being the root of the tree (see Figure 6.3).

The topology tree nodes are implemented by the `Manager.TopologyTreeNode` class and provide the following information:

- the subnet’s name;
- the names of hosts and routers physically connected to this subnet;

- a flag indicating the presence of an active MDM on this subnet;
- the number n_l of local active hosts on this subnet;
- the number n_s of active hosts on the subnet's 'subtree' (the term subtree here denotes the set of subnets located in hierarchically lower levels in the topology tree, including the present subnet itself), hence $n_s \geq n_l$;
- pointers to the upper level tree nodes[1];
- pointers to the next level tree nodes;
- a list of graphical components, each corresponding to a specific host, that will be made visible on the topology map upon discovering an active MAS entity on that host.

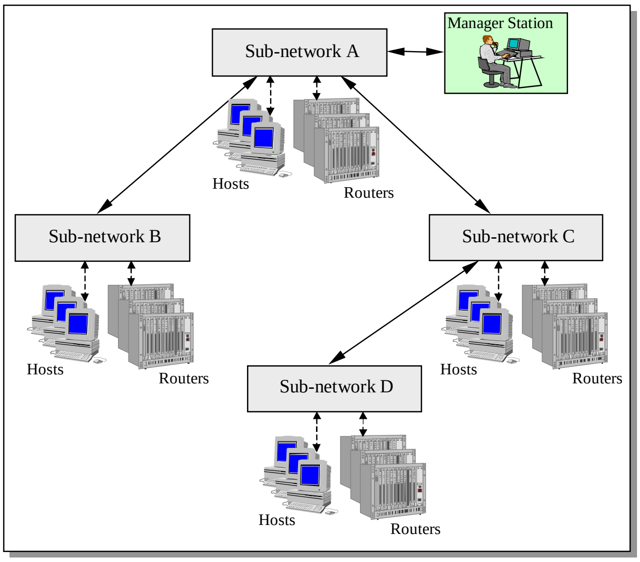


Figure 6.3. The topology tree structure

For instance, the number of active hosts in the subtree of Subnet A (in Figure 6.3) will be:

$$n_{s,subA} = n_{l,subA} + n_{s,subB} + n_{s,subC} = n_{l,subA} + n_{l,subB} + n_{l,subC} + n_{l,subD} \qquad (6\text{-}1)$$

All the information related to the managed network described above, is given to the manager application upon its initialisation, through parsing the "*network configuration*" file, described in Section 4.4.1.1. That file does not include activity status information, which is automatically discovered. For each file entry, a new `TopologyTreeNode` is created and inserted into the topology tree. In particular, its 'parent' (upper-level) subnets are located and then the next-level pointer of the parent nodes as well as the upper and next-level pointers of the inserted node are updated.

[1] A `TopologyTreeNode` may have more than one 'parent' nodes. This is certainly inconsistent with the definition of tree structures (as found in graph theory textbooks), hence, the use of the term 'tree' herein is abusive.

As discussed in the following sections, the topology tree plays a crucial role when the manager application needs to make a decision on which subnets require the deployment of an MDM entity.

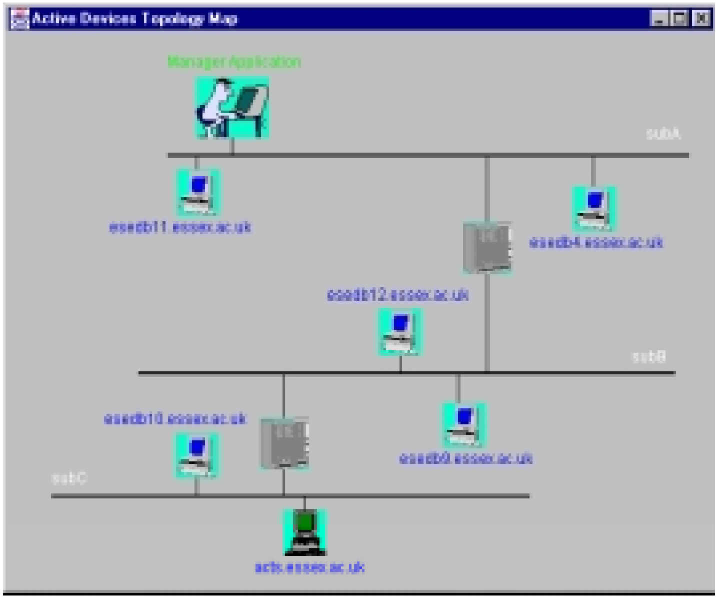


Figure 6.4. Topology map GUI

The topology tree is visually represented by the *topology map* Graphical User Interface (GUI), implemented by the `GUIs.TopologyMap` class; that is a graphical component of the manager application used to view the devices with currently active MAS servers. Managed devices are graphically illustrated with different icons depending on their hardware platform (PCs, workstations, etc). The topology map is updated at real-time when a new MAS is initialised or an existing one terminates. Such events are detected by the manager's Network Discovery thread, as discussed in Section 4.4.1.1. A snapshot of the topology map is shown in Figure 6.4.

6.3.2. MDMs implementation

The functionality of MDM entities is defined in the `MCode.MDM` class. As already mentioned, MDMs are essentially *mobile objects*. Namely, the MDM class extends the `MCode.MA` 'superclass', which defines the basic properties required to implement mobility characteristics (see Section 4.4.1.2). Yet, MDMs also act as remote *managers*. However, due to a limitation of Java programming language which does not support multiple inheritance, i.e. it does not allow a Java class to subclass two separate 'parent' classes, `MDM` class cannot extend the `Manager.Manager` class. For that reason, the `Manager.ManagerInterface` has been implemented. As its name suggests, ManagerInterface is a Java *interface*[2] including the definitions of a number of *abstract*[3] methods. Both Manager and MDM classes implement the ManagerInterface in addition to the java.util.Observer interface, which allows them to receive events notifications from the MAG tool[4], when an existing Polling Thread (PT) configuration is modified by the administrator.

MDMs deployment involves the transfer of the MCode.MDM class and also a number of classes that assist the MDM in performing its decentralised NSM operations. These include:

- the Manager.PT class, which controls the execution of specialised monitoring tasks already defined by the administrator and the Manager.PTC class, whose instances comprise descriptions (configurations) of the PTs;
- the MCode.DomainMonitoring and MCode.RI class which, as described in Section 6.3.6, are responsible for checking the utilisation of the hosts located within the MDM's domain and ensuring that the MDM executes at the least loaded host;
- the RMI.MdmRmiServer class, implementing the MDM’s RMI server which enables the communication between the manager and the MDM.

All these *six* classes are packaged in a *jar*[5] file whose size is 23.4 KB. The jar file is sent along with an MDM object carrying information about its assigned domain (e.g. the names of the managed devices located therein) and descriptions of the management operations it will perform. Upon arriving to a remote host, the MDM first starts its RMI server and then *registers* itself to the manager application (the registerMDMToMAG() method of the ManagerRmiServer class is invoked). The registration process is necessary in order to let the manager know the MDM’s current location and also to subscribe to events generated by the MAG tool whenever a PT configuration is modified. MDM’s registration process comprises two phases: first, the MDM passes its current location IP address to the

[2] *Interfaces* are defined similarly to classes, but include only declarations of methods. The designer of an interface declares the methods that should be supported by classes that *implement* the interface and what the functionality of these methods should be. Interfaces represent a powerful mechanism that helps to achieve many of the advantages of multiple inheritance, without its problems [ARN96].

[3] Methods declared as *abstract* can be defined either in abstract classes or interfaces. Their implementation is not supplied; the classes extending the abstract class or implementing the interface are enforced to provide specific implementations of the abstract methods [ARN96].

[4] The Manager.MAG class extends the java.util.Observable class. An *observable* object can have one or more *observers* (objects implementing the java.util.Observer interface). When an observable instance changes, the Observable's notifyObservers() method is called, causing all its observers to be notified of the change by invoking their update() method [ARN96].

[5] The *Java Archive* (JAR) file format enables bundling multiple files into a single archive file. Typically, a JAR file contains the class files and auxiliary resources associated with applets and applications. The JAR file format provides many benefits, namely security, decreased download time, compression, portability, etc [JAR].

ManagerRmiServer object. The latter will then use that address to obtain (again through RMI) a *reference* of the MDM object, which in turn is added to the MAG's list of observers[6]. When an event is generated (a PT configuration is updated), the MAG will automatically propagate the event to all the observers (i.e. the manager application and the remote MDMs), which in turn will adjust the operation of the updated PT. The event propagation is completely transparent to the user, who does not have to be aware of the remote location where MDMs execute.

Upon successful registration, the MDM instantiates the DomainMonitoring object and starts the execution of the PT threads. Likewise, before migrating to another host, the MDM first enforces executing PTs to return their collected data to the manager, suspends the execution of the PTs and unregisters itself from receiving MAG's events (through invoking the removeMDMFromMAG() method of the ManagerRmiServer class).

6.3.3. MDMs Deployment Policies

A key characteristic of our hierarchical framework is its dynamic adaptation to changes on the managed network. The structure of the proposed model is not rigidly configured, as MDMs may be dynamically deployed to specific network domains, given that certain requirements are met. Specifically, the administrator may explicitly set (through the GUI shown in Figure 6.5) the policies that define the hierarchical NMS operation, i.e. specify the criteria that should be satisfied for deploying an MDM to a network segment. The settings customising the operation of the hierarchical management model are stored in an instance of the Manager.HierarchicalSettings class.

In general, the deployment of MDMs may conform to either of the two following policies:

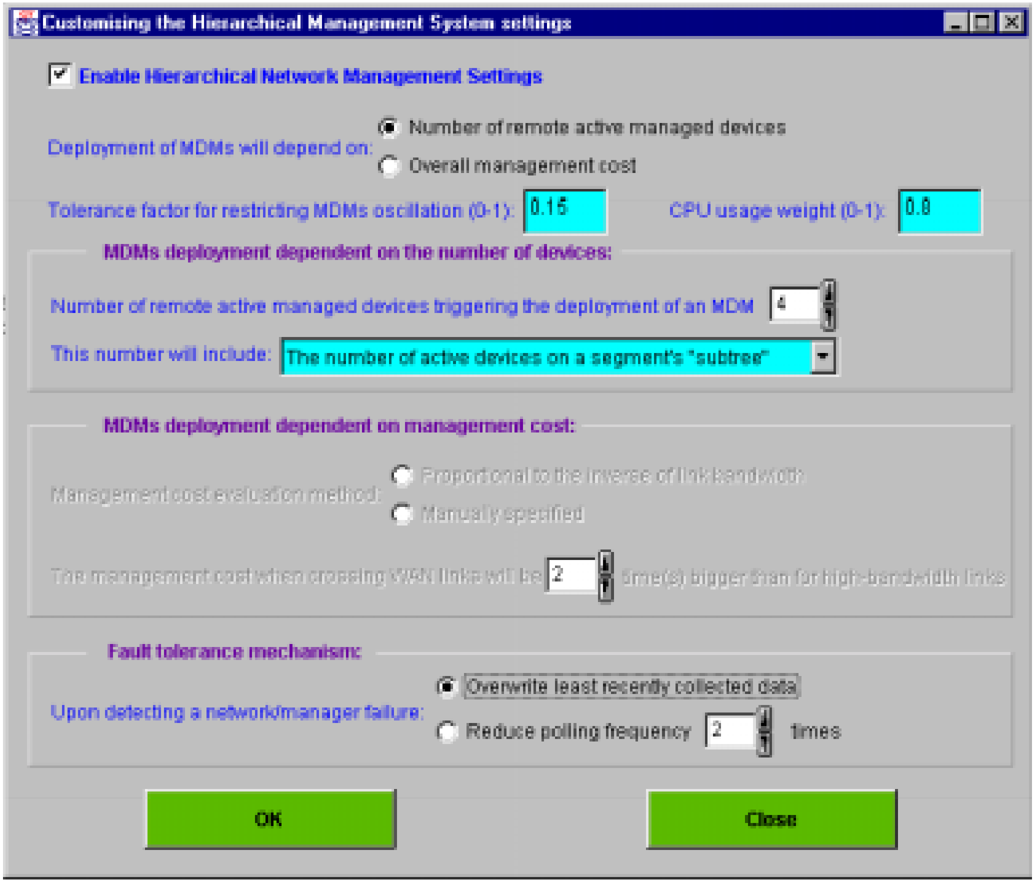
- **Policy 1**: the population of remotely active managed devices.
- **Policy 2**: the overall cost involved with the management of a remote set of devices.

When applying *Policy 1,* the administrator specifies the number of remote managed NEs that will justify the deployment of an MDM to a particular network segment. This number may either denote n_l or n_s (see Section 6.3.1). If, for instance, the specified number N denotes the

[6] Dividing the registration process in two phases was necessary. Namely, it would not be possible to simply pass an instance of an MDM object and append it to the observers' list. That would presuppose that the PT objects controlled by the MDM would be serialised (when a method passes an object's instance through RMI, the object is actually serialised and so are the objects referenced by the serialised object). However, that would not be feasible as Java threads cannot be serialised. To get around this problem, the registerMDMToMAG() method obtains a reference of the MDM object. It is the MDM's reference that will be added to the MAG's observers list. Thus, when a MAG event is generated, the MDM's update() method will be basically executed at the remote host where the MDM is running and the PTs vector is maintained. Through this programming trick, i.e. by registering a *reference* and not an *instance* of the remote MDM, we obviate the need for serialising the MDM object.

population of the examined subnet's local devices n_l, an MDM will be deployed to every network segment S with $n_{l,S} \geq N$; the boundaries of the domain assigned to the MDM will then be limited to that segment. If, on the other hand, N denotes the active hosts on the subnet's 'subtree' n_s, the domain assigned to the MDM will include all the active hosts located within the examined subnet's subtree, excluding the hosts already assigned to another MDM. The MDM will initially migrate to the least loaded host included into its assigned domain (this issue is discussed in detail in Section 6.3.6).

When the population of NEs directly managed by an MDM exceeds a certain limit, that domain will be divided to two independent domains, with another MDM undertaking the management of the second domain. In particular, the remote MDM will be instructed to create a clone of itself (through invoking the `clone()` method of the `java.lang.Object` class). The original MDM will pass to its clone information regarding its assigned domain, before dispatching it to a new location. The cloning approach is preferable than deploying a new MDM from a central location, as it reduces the deployment time in addition to saving the network overhead associated with the deployment process.



Customising the Hierarchical Management System settings

☑ Enable Hierarchical Network Management Settings

Deployment of MDMs will depend on: ● Number of remote active managed devices ○ Overall management cost

Tolerance factor for restricting MDMs oscillation (0-1): 0.15

CPU usage weight (0-1): 0.8

MDMs deployment dependent on the number of devices:

Number of remote active managed devices triggering the deployment of an MDM: 4

This number will include: The number of active devices on a segment's "subtree"

MDMs deployment dependent on management cost:

Management cost evaluation method: ○ Proportional to the inverse of link bandwidth ○ Manually specified

The management cost when crossing WAN links will be 2 time(s) bigger than for high-bandwidth links

Fault tolerance mechanism:

Upon detecting a network/manager failure: ● Overwrite least recently collected data ○ Reduce polling frequency 2 times

Figure 6.5. GUI for customising the hierarchical NMS policies

Accordingly, when the number of NEs managed by an MDM is reduced so that the requirements that originally triggered the MDM's deployment are no longer met, the manager application might consider of *merging* two or more management domains. Specifically, it will request an MDM to assign its managed devices to a neighbour MDM and subsequently terminate its execution.

When applying *Policy 2*, the management cost may either be: (a) proportional to the inverse of link bandwidth (a link with low bandwidth implies higher cost than a high-speed link), or (b) manually specified. It should be noted that Policy 2 is not implemented in the current prototype.

6.3.4. MDMs Deployment Implementation

Upon discovering an active MAS module, the topology tree is scanned to locate the corresponding host and the subnet where the host resides; the host's icon is also instantly made visible on the topology map. Following that, the number n_l of active hosts on that subnet is increased by one and subsequently, through following the pointer to the upper-level nodes, all the topology tree nodes up to the root are traversed and their number n_s of subtree nodes is also updated (increased by one). A similar procedure is followed when a MAS server is being shut down.

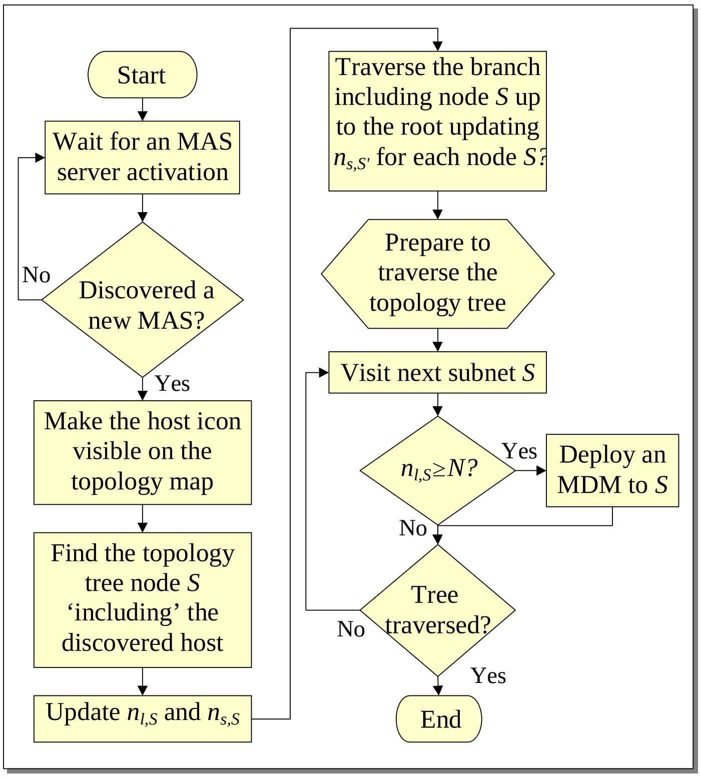
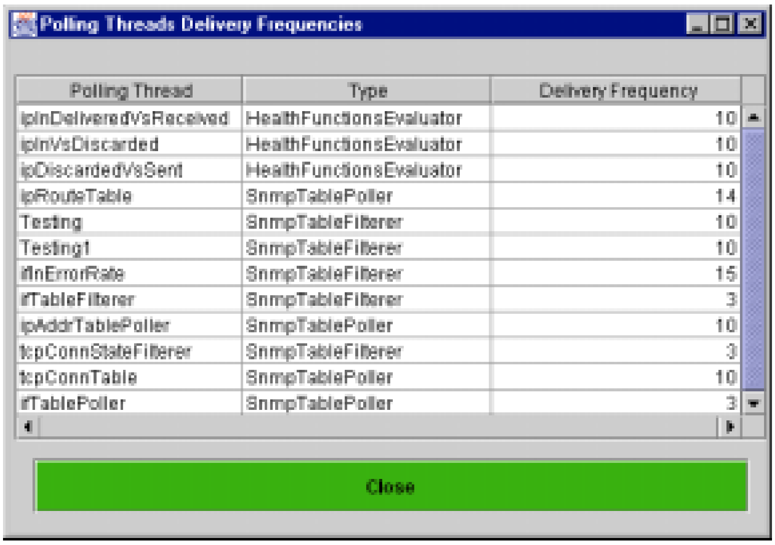

Start
Wait for an MAS server activation
Discovered a new MAS?
No
Yes
Make the host icon visible on the topology map
Find the topology tree node S 'including' the discovered host
Update $n_{l,S}$ and $n_{s,S}$
Traverse the branch including node S up to the root updating $n_{s,S'}$ for each node S'
Prepare to traverse the topology tree
Visit next subnet S
$n_{l,S} \geq N$?
Yes
Deploy an MDM to S
No
Tree traversed?
No
Yes
End

Figure 6.6. MDMs deployment algorithm (an MDM is deployed to each subnet including at least *N* active MAS servers).

The discovery or termination of a MAS server triggers an event at the manager host. The topology tree is then scanned with the subnets that meet certain requirements (determined by the `HierarchicalSettings` object) added to a list. In case that 'Policy 1' is employed, referring to the policies listed in the previous section, that list will include the subnets with n_l or n_s (depending on whether the MDMs deployment is a function of the active devices running locally or in the whole subtree) greater or equal to the specified constant N. If 'Policy 2' was employed, the cost corresponding to the management of each subnet would be evaluated and the list of subnets created accordingly. Ultimately, an MDM will be deployed to each of the subnets included in the list. The MDM deployment algorithm is illustrated in the block diagram of Figure 6.6, for the simple case where an MDM is deployed to each subnet including at least N active MAS servers.



Polling Threads Delivery Frequencies

Polling Thread	Type	Delivery Frequency
ipInDeliveredVsReceived	HealthFunctionsEvaluator	10
ipInVsDiscarded	HealthFunctionsEvaluator	10
ipDiscardedVsSent	HealthFunctionsEvaluator	10
ipRouteTable	SnmpTablePoller	14
Testing	SnmpTableFilterer	10
Testing1	SnmpTableFilterer	10
ifInErrorRate	SnmpTableFilterer	15
ifTableFilterer	SnmpTableFilterer	3
ipAddrTablePoller	SnmpTablePoller	10
tcpConnStateFilterer	SnmpTableFilterer	3
tcpConnTable	SnmpTablePoller	10
ifTablePoller	SnmpTablePoller	3

Figure 6.7. Adjusting PTs delivery frequency

Certainly, the set of management tasks already performed by the manager on these subnets will need to be conveyed to the MDM deployed therein. This is achieved through sending existing Polling Thread Configurations (PTC) along with the MDM. PTCs functionality has been described in Section 5.2.1.1. The reason for uploading the PTCs rather than the actual PTs is that the latter cannot be transparently transferred along with the MDM retaining their execution state, as Java does not support threads serialisation/deserialisation[7]. On the other hand, PTC objects are serialisable. Upon its arrival at the remote subnet, the MDM instantiates the PTs using their corresponding configurations, encapsulated into the MDM's state. PTs will thereafter start performing their tasks without any further disruption of the management process. Individual PTs are responsible for periodically delivering their collected data to the manager. The delivery frequency of monitoring tasks can be remotely adjusted by the

[7] The `java.lang.Thread` class does not implement the `java.io.Serializable` interface.

administrator, through the GUI shown in Figure 6.7; changes in delivery frequencies are communicated to the PTs through invoking the `update()` method of the `MdmRmiServer` class. It is noted that PTs are not aware of whether they operate under the control of the manager application or an MDM, as both `Manager` and `MDM` classes implement the `ManagerInterface`. That enables a flexible design and maximises code reuse.

6.3.5. MA Code Distribution Scheme

A key feature of MA-based NSM frameworks is their ability to dynamically customise management services. Our hierarchical model should therefore ensure that new NSM services (implemented by specialised MAs) can be introduced/updated at runtime, even post MDMs deployment. However, that should not affect the framework's performance, which primarily depends on the size of travelling MA entities. In a typical Java-based MAP, both the MA *code* and *state* are required at the destination to instantiate the received MA objects. Nicklisch et al. [NIC98] identified three alternative agent code transfer schemes: (a) "*push*", (b) "*pull*" and (c) "*migrate*" (see Section 3.6.1.3).

The majority of MAPs proposed for management applications [KU97, ZAP97, COR98a, BEL99, SAH98, SUS98], with the exception of [SOA99] and [PUL00a], involve transfer of both the MA's code and state on each migration, namely they employ the "migrate" approach. In Section 4.4.1.4.1, we have described a code distribution scheme implementing the "push" scheme; i.e. bytecode is distributed at the MA's construction time with only the state information transferred thereafter, resulting in a much lower demand on network resources. The "migrate" code distribution scheme offers a better starting point in terms of the associated network overhead, since the bootstrapping procedure described above is not required. However, it is outperformed by the "push" approach, after a small number of PIs elapses.

The introduction of the hierarchical management model reduces the code distribution cost even further by adopting a "*tree multicasting*" approach. In particular, MA bytecode distribution takes place in two successive phases. In the first phase the MAs bytecode is no longer multicasted to all managed devices as in the flat model (see Section 4.4.1.4.1.), but instead distributed to the MAS entities local to the manager segment and also to all active MDMs. The latter will then forward the received bytecode to their supervised NEs (see Figure 6.8). Should a remote domain including *N* hosts connects to the manager site through a low-bandwidth link, the tree multicasting approach will considerably decrease management cost as bytecode is transferred only once (instead of *N* times) through the interconnecting link. The execution of the monitoring task implemented by the uploaded MA class is started immediately by a dedicated PT, instantiated by the MDM for this purpose.

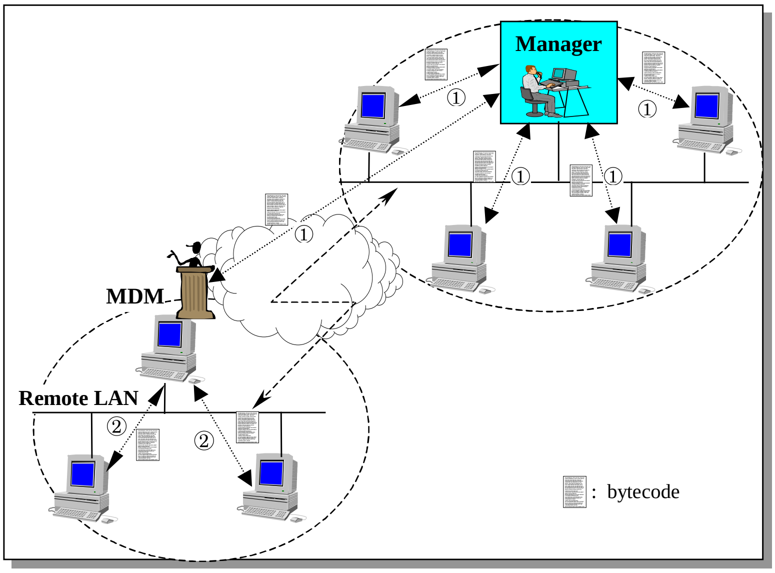


Figure 6.8. MA bytecode through “tree multicasting”

6.3.6. Processing Load Balancing

Although MDMs have been designed to be as lightweight as possible, they cannot avoid consuming memory and processing resources on the NE where they execute. The framework should therefore be sufficiently flexible to allow MDMs to autonomously move to another host, when their current hosting device is overloaded, to provide more balanced distribution of the overall processing load.

This is accomplished through the regular inspection of the domain’s NEs, in terms of their memory and CPU utilisation: an MA object, termed *Resource Inspector* (RI), implemented by the `MCode.RI` class, is periodically dispatched (by the `DomainMonitoring` thread) and visits all the local devices obtaining utilisation figures before delivering the results to the MDM. Host load figures represent their average load over relatively long time windows to avoid sensitivity to temporal utilisation peaks. If the hosting system is seriously overloaded, compared to its neighbouring devices, the MDM will transparently move to the least loaded node. In the example depicted in Figure 6.9, a RI object sequentially visits all the managed devices in the MDM's local domain. At each host, the RI obtains the average CPU & memory load values during the last interval, keeping track of the least loaded device (in this example that will be Host *D*). Finally, the RI reports its results to the MDM, which in turn transparently migrates from Host *A* to Host *D* after informing the manager application about its decision. The correct and reliable operation of the MDMs transparent location change has been verified by overloading an MDM’s hosting device (through launching several memory-consuming applications and running heavy background jobs), thereby enforcing the MDM to migrate to a least loaded host.

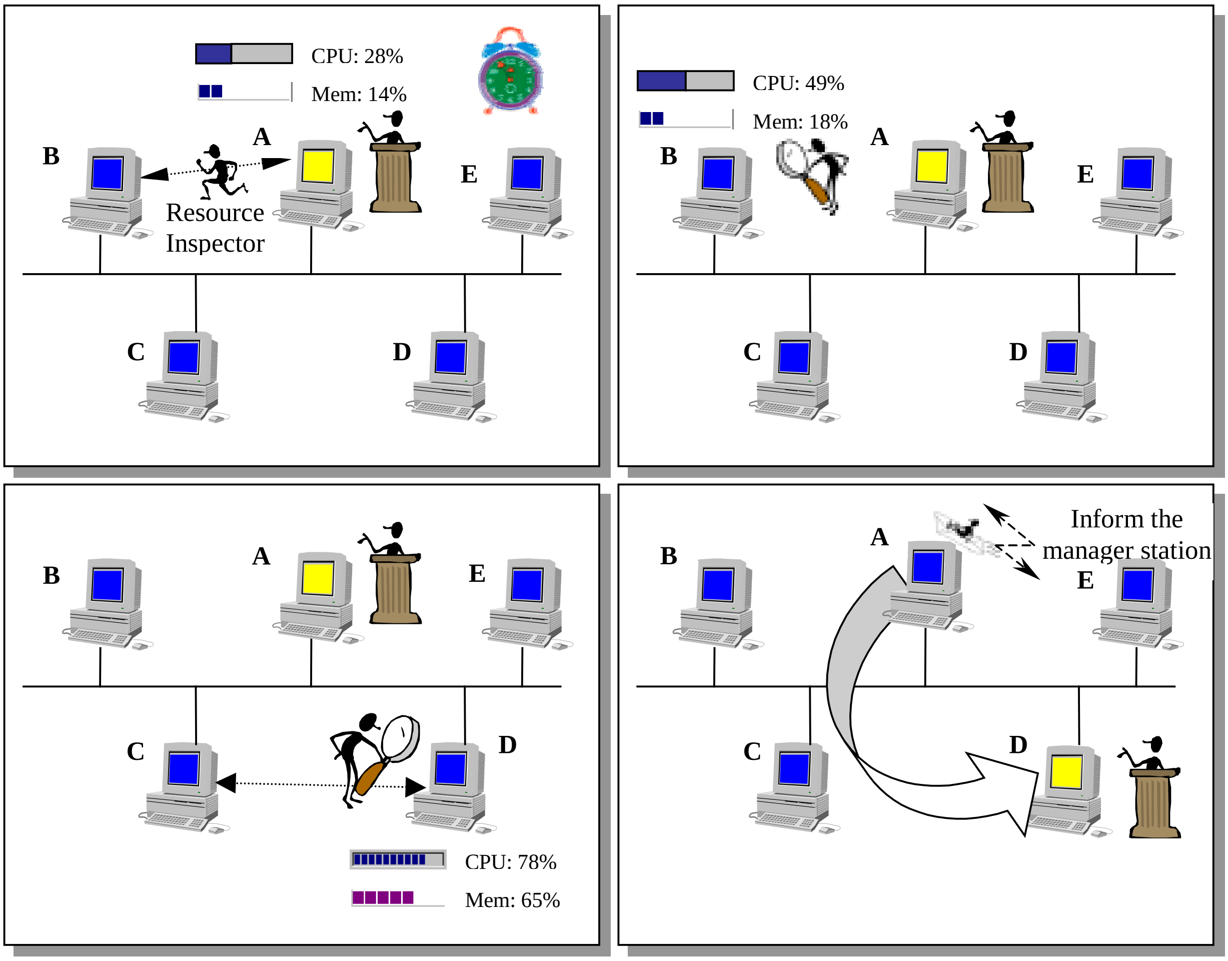


Figure 6.9. Obtaining host load profiles by a Resources Inspector MA object

MDMs are prevented from continuously *oscillating* between different hosts through adjusting the value of the *tolerance factor*, 0< t_f < 1. Assuming that an MDM executes on a host *x*, with average utilisation U_x, the MDM will consider migrating to another host *y* only if its utilisation is $U_y < (1 - t_f)\, U_x$. If for instance, $t_f = 0.2$, the MDM will not migrate to *y* unless its utilisation level is at least 20% lower than that of *x*.

A host's average utilisation U_x is a linear function of the CPU and memory usage, i.e. $U_x = a \times CPU_x + b \times M_x$. In particular, the administrator determines through the GUI shown in Figure 6.5 the weight *a* that CPU usage CPU_x will have on the evaluation of the aggregate host utilisation, where $0 < a < 1$. The weight of memory usage M_x will then be: $b = 1 - a$.

The MDM notifies the manager application about its new execution location before the actual migration occurs. Should the manager attempt to contact the MDM while the latter is still moving, an exception will be thrown and a new attempt to contact the MDM will take place after a specified interval; in the meanwhile, the MDM arrives at its new hosting device and is capable of receiving manager's messages. The location where the MDM executes is indicated on the manager's topology map through a label displayed next to the MDM's hosting device icon. Location changes trigger real-time map updates. For instance, Figure 6.10 illustrates two snapshots of the topology map before and after the migration of an MDM from esedb9.essex.ac.uk to esedb12.essex.ac.uk host.

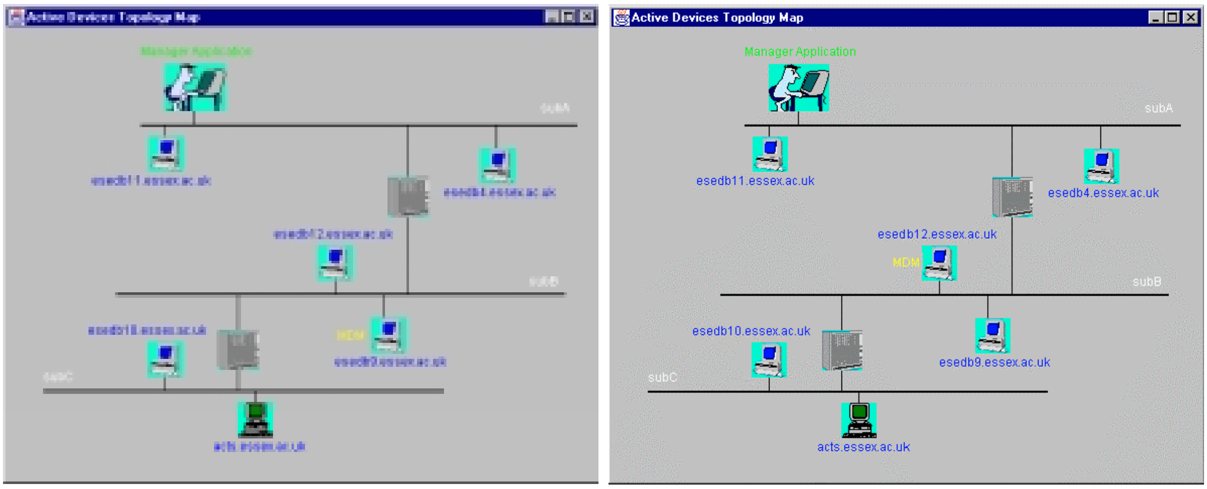


Figure 6.10. Illustration of an MDM location change on the manager's topology map

6.3.7. Resources Monitoring Tool Implementation

In order to obtain devices' load reports we have built a Resources Monitoring Tool (RMT), developed in C, able to accurately measure the CPU and memory usage. On Windows platforms, low-level functions of the Win32 API[8] [Win32] have been utilised, whereas standard Unix commands are executed under Solaris.

The integration of this tool with the MAS application, developed in Java, is achieved through the Java Native Interface (JNI), discussed in Section 2.7.1.1. The JNI allows Java code running within a JVM to inter-operate with applications and libraries written in other languages, such as C or C++. JNI is used to write native methods to handle situations where an application cannot be written entirely in Java. The steps required to achieve the integration of Java and native programs are described in [JNI]. The Java front-end (methods) accessed by incoming RIs provides them a uniform interface onto the local resources, whilst hiding the underlying architecture, i.e. the native methods implementation.

Snapshots of hosts' load profiles are taken in regular intervals. The duration of these intervals, termed *observation periods* O_p, should be carefully specified: O_p should be long enough to avoid sensitivity on sporadic load peaks and, at the same time, short enough so as not to omit potentially prolonged increments of processing load.

[8] The Microsoft Win32 API provides building blocks used by applications written for Microsoft Windows NT, Windows 2000, and Windows 95/98. Among others, an application using Win32 functions is able to manage system objects such as memory, disks, files, and processes.

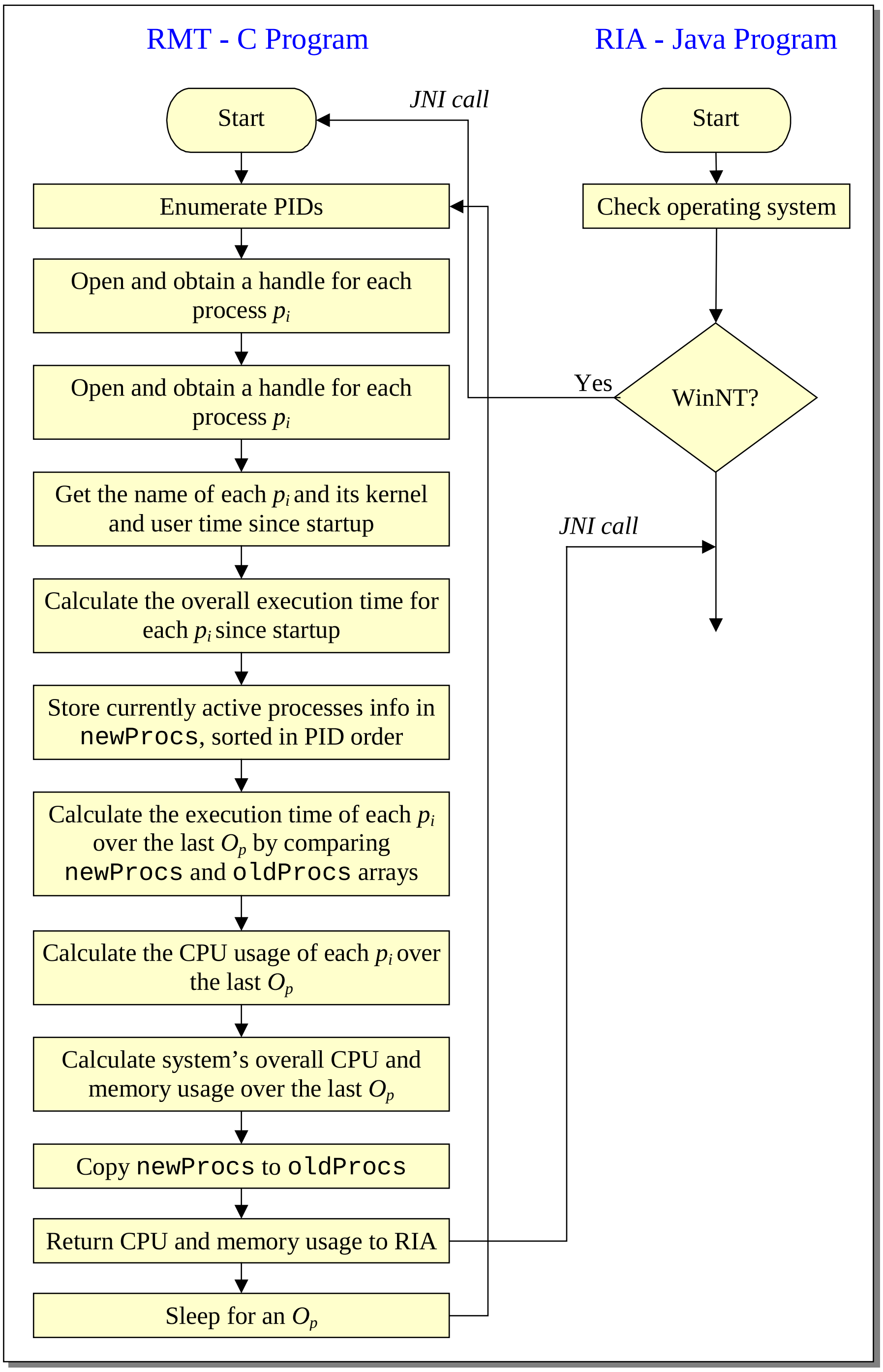

RMT - C Program
RIA - Java Program
Start
JNI call
Start
Enumerate PIDs
Check operating system
Open and obtain a handle for each process p_i
Open and obtain a handle for each process p_i
Yes
WinNT?
Get the name of each p_i and its kernel and user time since startup
JNI call
Calculate the overall execution time for each p_i since startup
Store currently active processes info in newProcs, sorted in PID order
Calculate the execution time of each p_i over the last O_p by comparing newProcs and oldProcs arrays
Calculate the CPU usage of each p_i over the last O_p
Calculate system's overall CPU and memory usage over the last O_p
Copy newProcs to oldProcs
Return CPU and memory usage to RIA
Sleep for an O_p

Figure 6.11. RMT and RIA operation

The operation of the RMT tool is only briefly described in the following (a detailed description would be beyond the scope of this thesis): RMT's execution is started and controlled by the Resource Inspection Application (RIA) of the MAS server (see Section 4.4.1.3.7), defined in the `MAS.RIA` class. A RIA thread is created and started upon the MAS initialisation. First, RIA checks the identity of the underlying operating system[9]. Should RIA runs on a Windows platform, the RMT tool is initialised through a native call and starts monitoring the local resources usage (in time intervals defined by the O_p).

In particular, RMT periodically executes the `EnumProcesses()` function [Win32], which returns an enumeration of active system process identifiers (PID). Then, for each retrieved PID, the `OpenProcess()` is invoked, opening and returning a handle to the process object p_i. Handles are used as process references to obtain the process name (`GetModuleBaseName()` function) and also the process creation and execution time, through calling `GetProcessTimes()` function. In particular, when invoking the latter function at instant t_n, it returns the time period that process p_i has spent executing in kernel K_{p_i,t_n} and user mode U_{p_i,t_n} since system's startup.

The overall process execution time T_{p_i,t_n} is then calculated ($T_{p_i,t_n} = K_{p_i,t_n} + U_{p_i,t_n}$). Individual process memory usages M_{p_i} are also found through calling the `GetProcessMemoryInfo()` function. The information referring to currently active processes is stored in `newProcs` array[10], sorted in PID order. The execution time of each individual process over the last O_p is then calculated, by using the information stored in `newProcs` and `oldProcs`[11] arrays: $T_{p_i,O_p} = T_{p_i,t_n} - T_{p_i,t_{n-1}}$. The CPU usage of p_i is then easily found: $CPU_{p_i,O_p} = {T_{p_i,O_p}} \big/ {O_p}$. The aggregate system and memory usage can be evaluated by summing up the individual process CPU and memory usages: $CPU_{O_p} = \sum_i CPU_{p_i,O_p}$, $M_{O_p} = \sum_i M_{p_i}$.

Finally, the contents of `newProcs` array are copied to `oldProcs` and RMT delivers a system resources report to RIA through a JNI call. RMT then 'sleeps' for a time period equal to O_p (the `Sleep()` function is called) before resuming execution and repeating the procedure described above. RMT and RIA operation is illustrated in the block diagram of Figure 6.11. An overview of the process enumeration procedure is given in [Win32Enum].

[9] This is achieved through simply examining the value of the local system's file separator (the `getProperty("file.separator")` method of the `java.lang.System` class is invoked). The file separator is '\' for Windows systems and '/' for Unix systems.

[10] `newProcs` array contains a set of records, each of which corresponds to a specific process p_i and includes its PID, its overall execution time since system's startup T_{p_i,t_n} and its memory usage M_{p_i}. The information stored in `newProcs` array refers to instant t_n.

[11] `oldProcs` array stores similar information to `newProcs`, but refers to instant t_{n-1}, where t_n-t_{n-1}=O_p.

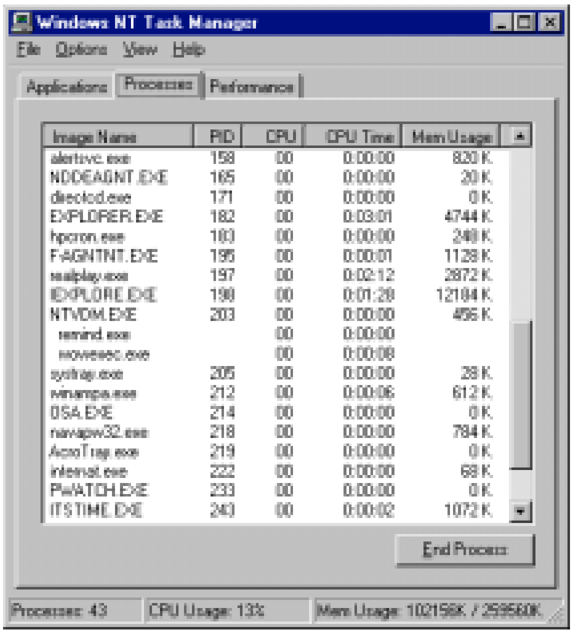


Image Name	PID	CPU	CPU Time	Mem Usage
alertsvc.exe	158	00	0:00:00	820 K
NDDEAGNT.EXE	165	00	0:00:00	20 K
directcd.exe	171	00	0:00:00	0 K
EXPLORER.EXE	182	00	0:03:01	4744 K
hpcron.exe	183	00	0:00:00	248 K
F-AGNTNT.EXE	195	00	0:00:01	1128 K
realplay.exe	197	00	0:02:12	2872 K
IEXPLORE.EXE	198	00	0:01:28	12184 K
NTVDM.EXE	203	00	0:00:00	456 K
remind.exe		00	0:00:00	
movexec.exe		00	0:00:08	
systray.exe	205	00	0:00:00	28 K
winampa.exe	212	00	0:00:06	612 K
OSA.EXE	214	00	0:00:00	0 K
navapw32.exe	218	00	0:00:00	784 K
AcroTray.exe	219	00	0:00:00	0 K
internat.exe	222	00	0:00:00	68 K
PWATCH.EXE	233	00	0:00:00	0 K
ITSTIME.EXE	240	00	0:00:02	1072 K

Processes: 43 CPU Usage: 13% Mem Usage: 102156K / 259560K

Figure 6.12. The Windows NT Task Manager

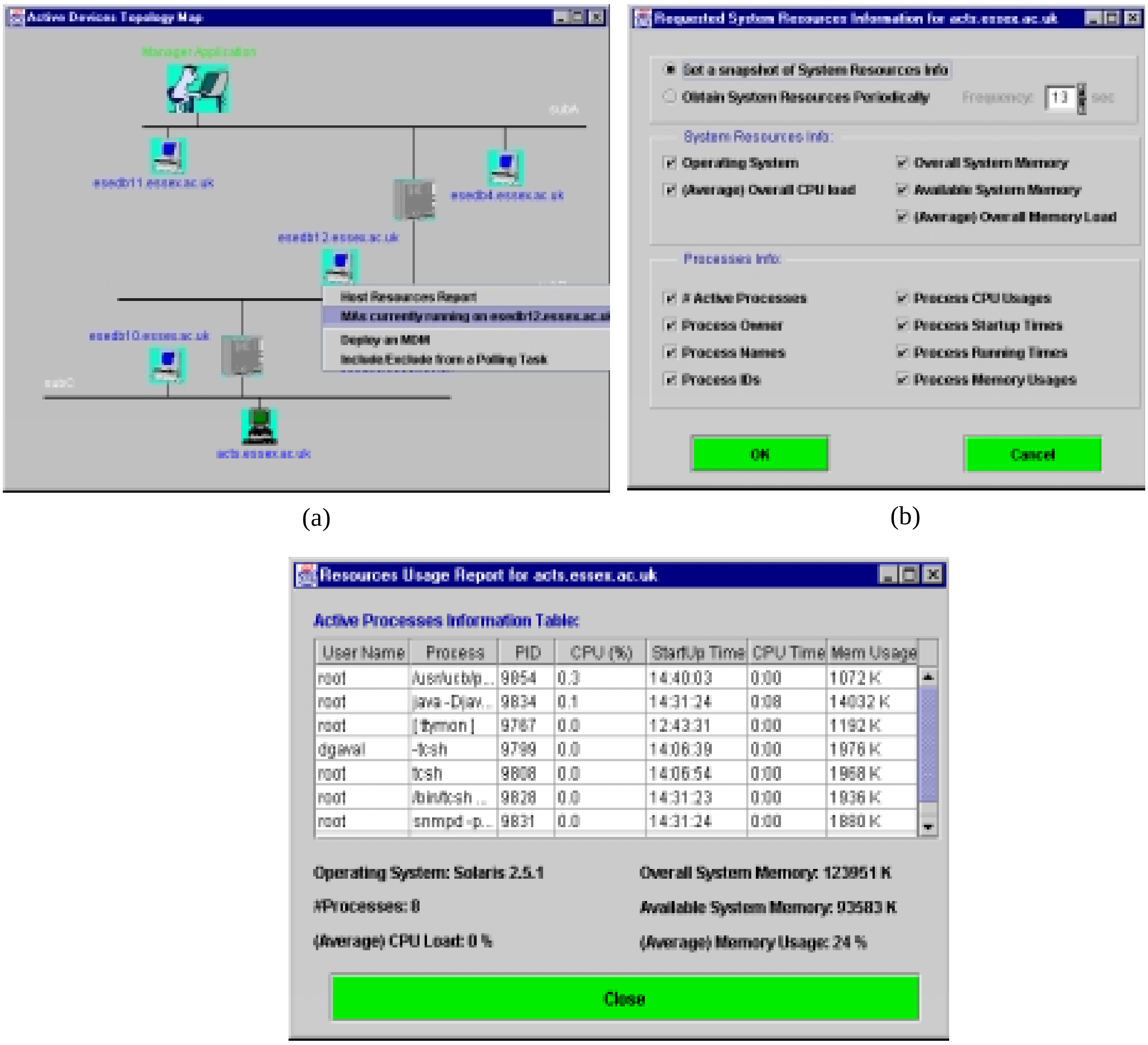
The C file implementing the procedure described above, along with the accompanying header files are compiled creating a library (DLL) file. That library, along with `psapi.dll` file which includes the implementations of the Win32 API functions comprise the RMT tool, representing an overall size of 100Kbytes. Among the existing Windows operating systems family, the current implementation of the RMT supports only Windows NT platforms; however, as claimed in [Win32Enum] only minor modifications are required to extend its support to Windows 95/98 and 2000 systems. In general, the RMT tool provides information very similar to that provided by the Windows NT Task Manager application (see Figure 6.12).

Regarding Solaris operating systems, the standard UNIX command *ps -au* is periodically executed[12]. That command prints the list of active system processes along with their PID, CPU and memory usage, startup time, etc. The command's output is captured and subsequently parsed in order to obtain the host's CPU and memory utilisation following a similar approach to the one described for Windows platforms.

In addition to assisting MDMs to locate the least loaded host within their management domain, RMT can also be utilised to provide user reports, describing the load profile of a remote host. This can be achieved through simply right-clicking on the host's icon on the topology map (Figure 6.13a). The administrator then selects the kind of system resources information he/she is interested in (Figure 6.13b). The user's request is subsequently sent to the

[12] Operating system-specific commands can be executed by a Java application through calling the `exec(<command>)` method of `java.lang.Runtime` class. `exec` method returns the process (`java.lang.Process`) handling the command's execution. The command's output can also be captured (through the `getInputStream()` method of `java.lang.Process`) and subsequently parsed.

remote MAS server (as a `Manager.ReportDescription` object). In particular, the `MasRmiServer`'s `getResourcesReport()` method is invoked (see Figure 4.14), with the returned information displayed in the GUI shown in Figure 6.13c.



(a)

Requested System Resources Information for acts.essex.ac.uk

● Get a snapshot of System Resources Info
○ Obtain System Resources Periodically   Frequency: 13 sec

System Resources Info:

☑ Operating System
☑ (Average) Overall CPU load
☑ Overall System Memory
☑ Available System Memory
☑ (Average) Overall Memory Load

Processes Info:

☑ # Active Processes
☑ Process Owner
☑ Process Names
☑ Process IDs
☑ Process CPU Usages
☑ Process Startup Times
☑ Process Running Times
☑ Process Memory Usages

(b)

Resources Usage Report for acts.essex.ac.uk

Active Processes Information Table:

User Name	Process	PID	CPU (%)	StartUp Time	CPU Time	Mem Usage
root	/usr/ucb/p...	9854	0.3	14:40:03	0:00	1072 K
root	java -Djav...	9834	0.1	14:31:24	0:08	14032 K
root	[ttymon]	9767	0.0	12:43:31	0:00	1192 K
dgawai	-tcsh	9799	0.0	14:06:39	0:00	1976 K
root	tcsh	9808	0.0	14:06:54	0:00	1968 K
root	/bin/tcsh ...	9828	0.0	14:31:23	0:00	1936 K
root	snmpd -p...	9831	0.0	14:31:24	0:00	1880 K

Operating System: Solaris 2.5.1
#Processes: 8
(Average) CPU Load: 0 %

Overall System Memory: 123951 K
Available System Memory: 93583 K
(Average) Memory Usage: 24 %

(c)

Figure 6.13. Obtaining on-line resource usage reports from remote devices

6.3.8. Manager-MDMs Communication

One of our framework's key advantages is that it considerably reduces the amount of information exchanged between the manager platform and the managed devices. This is due to the introduction of the intermediate management level, realised by dynamically deployed MDMs. However, that does not obviate the necessity for bi-directional communication between MDMs and the manager host. Java RMI [RMI] has been chosen for implementing the communication bus between the distributed MDMs and the manager host, due to its inherent simplicity and the rapid prototype development that it offers.

In particular, the `RMI.ManagerRmiServer` class is used to pass information to the manager, while the `RMI.MdmRmiServer` class has been implemented to enable communication in the reverse direction (see Figure 6.14 for an overview of its methods). In the MDM-to-Manager direction, we have the following information flow:

- delivery of data reports generated through filtering raw data collected from local devices (these data have been obtained by MAs launched by the MDM's PTs);
- registration of the MDM to MAG's events;
- notification of the manager prior to an MDM's migration to another host.

```
public boolean terminate () {}  // Requests the MDM to terminate its execution
public boolean move (String host) {}   // Requests the MDM to migrate to another location
public boolean clone() {} /* Requests the MDM to create a clone and share the management
                              responsibility of its assigned domain with it */
public boolean deliverData () {}   // Requests the delivery of collected data from all running PTs
public boolean addPT (PTC configuration) {} // Uploads a new PT configuration
public boolean removePT (String PTname) {} // Requests the disposal of an existing PT
public boolean update (String PTname, PTC newConfiguration) {}  /* Updates an existing PT
                                                                   configuration */
public boolean addManagedDevice (String host) {}  /* Add a new host to the MDM's list of
                                                     managed devices */
public boolean removeManagedDevice (String host) {}   /* Removes a host from the MDM's
                                                         list of managed devices */
public Vector getManagedDevices () {}  // Returns the MDM's list of managed devices
public void returnDataFolder (String PTname, String MaSeqNum, Vector data) {}  /* Invoked by an
                              MA with sequence number MaSeqNum, which returns its collected data
                              to the PT that originally launched it */
```

Figure 6.14. The methods of the RMI.MdmRmiServer class

On the opposite direction, the manager may:

- request the MDM to terminate its execution, move to another domain or create a clone of itself to share the management responsibility of its assigned domain;
- request the MDM to enforce its controlled PTs to deliver all their collected data;
- upload at runtime additional management services (PTCs) or request the termination of an existing one;
- update an existing PT configuration;
- add/remove a NE from the MDM's list of managed devices;
- obtain the list of the MDM's supervised devices.

6.3.9. Fault Tolerance

A key motivation for the development of the introduced hierarchical model has been to minimise dependency on network resources and exploit MAs ability of acting autonomously, without the manager's intervention. Hence, MDM entities have been designed so as to tolerate failures on the interconnecting links between their local subnets and the manager's site or on the manager platform itself. Such failures are typically detected when MDMs attempt to return aggregated results back to the manager. Upon detecting a failure (an exception is thrown when the data transfer cannot be completed), they continue to perform their decentralised tasks as normal while periodically checking for the status of the link and/or the manager. As soon as the communication is restored, all management data collected in the meanwhile are returned to the manager.

Certainly, in case that a large number of monitoring tasks are controlled by the MDM and should the interruption of the normal communication flow between the MDM and the manager be prolonged, the MDM's size will significantly grow. That may in turn have a serious impact on the MDM's hosting device resources. Hence, in order to maintain control over the growth of the MDMs state, the administrator may choose (through the GUI shown in Figure 6.5) between the following models: As soon as an MDM detects the failure it will either (i) overwrite the least recently collected management data by keeping only the latest acquired values, or (ii) reduce the polling frequency of the individual monitoring tasks so that the management data accumulation rate will decrease. The reliable operation of the fault tolerance mechanism has been verified by creating 'artificial' failures on the manager host, i.e. by shutting down the manager application and restarting it after an MDM detected the failure.

Future extensions will also cater for faults on MDMs hosting devices: The manager will periodically check the status of the devices where MDMs currently execute. Should a hosting device fails, the manager will automatically deploy another MDM to a 'healthy' host to take over the management of the remote subnet.

6.4. Quantitative Evaluation

This section presents mathematical formulations, which quantify the network overhead associated with the employment of the proposed hierarchical model. This quantitative evaluation builds upon previous evaluations, presented in Sections 4.5 and 5.3, using the same definitions, symbols and formalisms.

Although mobility can often be beneficial for NSM, overheads induced by MAs and MDMs in particular, e.g. due to their deployment and management should be accounted for very carefully. Slightly different configurations for a set of MDMs may result in dramatically variant network loads [LIO98]. Hence, it is important to define concrete cost functions estimating the corresponding overheads.

In this context, let the *cost coefficients* k_{S_i,S_j} denote the cost of sending a byte of information between arbitrarily indexed subnets S_i and S_j, where S_0 is the manager host location. For multi-hop connections, the cost coefficients will be equal to the summation of the individual links coefficients. In the following investigation, we make the simplifying assumption that an MDM may manage only the hosts included in a *single* subnet and not a wider set of devices.

A simple function characterising the bandwidth consumption for our hierarchical architecture, is the following:

$$C_{hier} = C_{distr} + C_{depl} + C_{pol} + C_{deliv} \qquad (6\text{-}2)$$

where the four terms represent the cost for distributing to the MAS servers the bytecode of the MA that will undertake the monitoring task, the MDMs deployment cost, the bandwidth used for the actual monitoring operation (polling) and the cost for delivering to the manager host the collected data, respectively.

Concerning bytecode distribution, the lightweight “tree multicasting” scheme described in Section 6.3.5 is adopted. The code distribution cost is therefore given by:

$$C_{distr} = \left(k_{S_0,S_0} * N_0 + \sum_{i=0}^{M} \left[k_{S_0,S^i} + k_{S^i,S^i} * \left(N_{S^i} - 1 \right) \right] \right) * C \qquad (6\text{-}3)$$

where M is the total number of active MDMs, C the compressed bytecode size, S^i represents the subnet including host i, and N_{S^i} the number of hosts included in subnet S^i.

Likewise, C_{depl} equals the cost of broadcasting M MDM objects to their corresponding remote domains:

$$C_{depl} = \sum_{i=0}^{M} k_{S_0,S^i} * (C_{MDM} + ST_0 + n * \overline{S}_{PTC}) \qquad (6\text{-}4)$$

where C_{MDM} is the size of the *jar* file uploaded at every MDM deployment[13] (= 23.4 Kb), ST_{i} represents the compressed state size of an MA when migrating from the i^{th} host and $\overline{S}_{PTC}$

[13] See Section 6.3.2.

the average size of each of the *n* PT configurations attached to the MDM (each serialised PTC object amounts approximately 250 bytes).

C_{pol} is defined as the summation of the cost induced for polling the NEs directly managed by the manager host and the cost associated with polling the NEs that operate under the MDMs control, multiplied with the number of PIs *p*:

$$C_{pol} = \left(\sum_{i=0}^{m} k_{S^i, S^{i+1}} * ST_i + \sum_{i=0}^{M} \sum_{j=0}^{N_i} k_{S^i, S^i} * ST_j \right) * p \qquad (6\text{-}5)$$

Clearly, the first term of the summation will dominate on the overall polling cost if the *m* devices managed by the central manager platform are spread among several subnets. Specifically, cost coefficients $k_{S^i, S^{i+1}}$ are typically larger when an MA migrates from subnet S^i to another subnet S^{i+1} $(S^i \neq S^{i+1})$ rather than when it moves within the same subnet $(S^i = S^{i+1})$. As discussed in Section 4.5.2, MAs state size ST_{i} does not remain constant, but increases for each visited node, with the increment rate depending on the selectivity σ, as shown by Eqn. (4.7). The last term appearing in Eqn. (6-2) represents the cost associated with the delivery of the gathered data from the MDMs to the manager host:

$$C_{deliv} = \frac{D * p}{F_{del}} * \sum_{i=0}^{M} k_{S^i, S_0} \qquad (6\text{-}6)$$

where F_{del} indicates (in number of PIs) the delivery frequency, namely how often MDMs package the computed statistics of size *D* and deliver them to the manager.

It should be emphasised that MDM functionality is not necessarily limited to simply gathering and delivering data to an upper-level manager. Although this thesis concentrates on data-intensive network monitoring applications, a broad spectrum of management applications (including fault, configuration and security management) could also be performed. Upon arriving at their remote domains, MDMs may autonomously make management decisions and take actions based on the values of collected MIB values (for instance when the value of an aggregation function of several MIB objects exceeds a pre-specified threshold). These actions may include first-line support to handle trivial faults, decisions to recalibrate the management system as a response to changes sensed to traffic patterns or network configuration, e.g. to share the management responsibility of its domain with another MDM, move to a nearby domain, terminate execution, etc.

6.5. Experimental Results

In order to evaluate the performance of the proposed hierarchical model, we have conducted a number of experiments, using the same testbed described in Section 5.4.2, i.e. a PC playing the role of the manager and 10 PCs with active MAS servers, simulating the managed devices. In the experiments described below, we make the assumption that the managed devices are separated from the manager platform by a WAN link (see Figure 6.15), although they all actually reside on the same network segment.

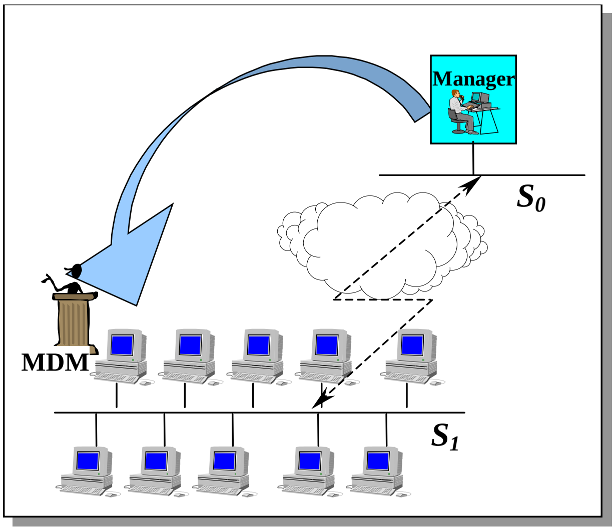
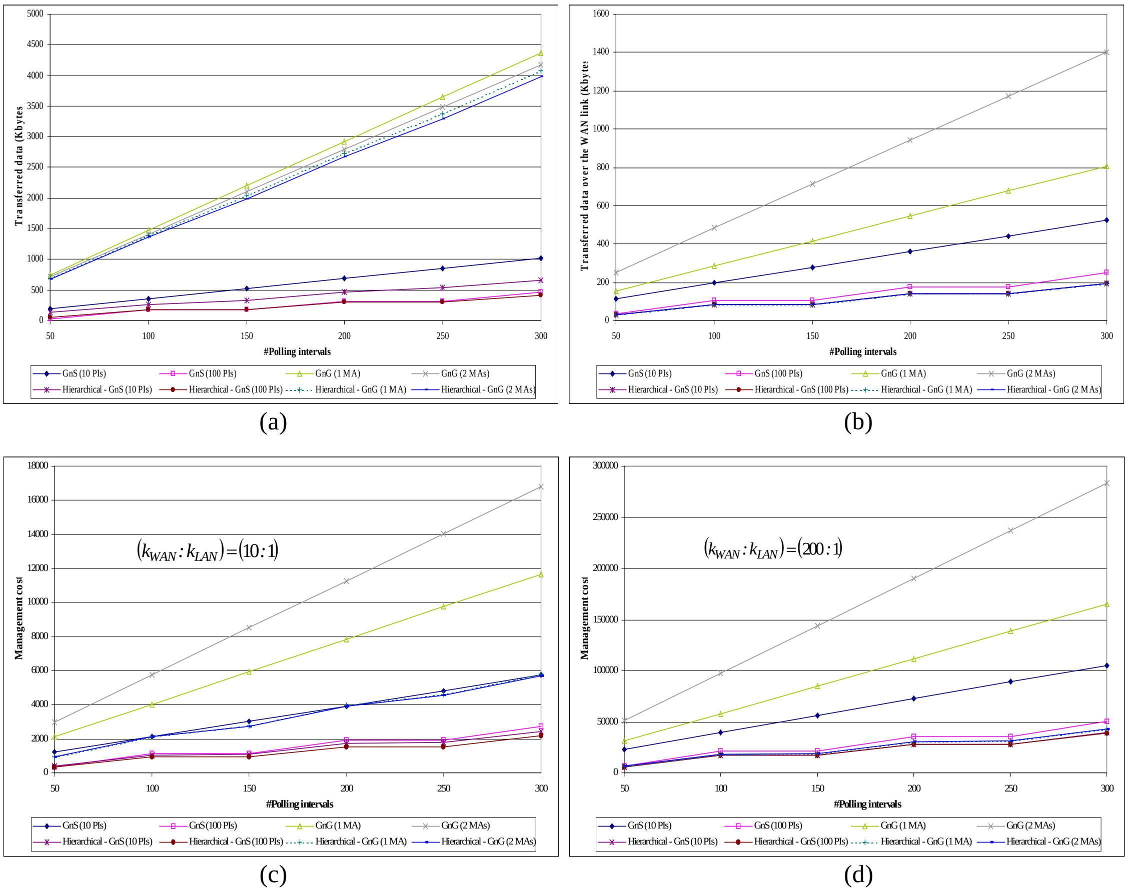


Figure 6.15. The experimental testbed

The examined scenario involves the collection of a data sample from each host at every PI; the size of each data sample can either be 50 (Figure 6.16) or 2000 bytes (Figure 6.17). We compare the performance of flat management, GnG (employing 2 MAs) and GnS polling (for data delivery frequencies of 10 or 100 PIs) against that of the hierarchical model. Regarding GnS polling, we have applied the *cloning* data delivery method described in Section 5.2.2.1, as that was proved the most efficient according to the experimental results reported in Section 5.4.2.

Referring to hierarchical management, a number of data collection approaches have been evaluated. In particular, the monitoring process is performed either through applying GnG (launching one or two MAs) or GnS polling (with data delivered every 10 or 100 PIs). Aggregated results are sent to the central manager with frequency of F_{del} = 100 PIs.

The deployment overhead associated with the MDM's accompanying *jar* file (23.4 Kb) has also been taken into account. The monitoring task is carried out by an MA object, whose definition (class file) amounts 2.2 KB. When employing, the GnG or GnS polling schemes, that file is multicasted to all active MAS entities, while in the hierarchical approach, the "tree multicasting" technique is applied (see Section 6.3.5).



(a)

(b)

(c)

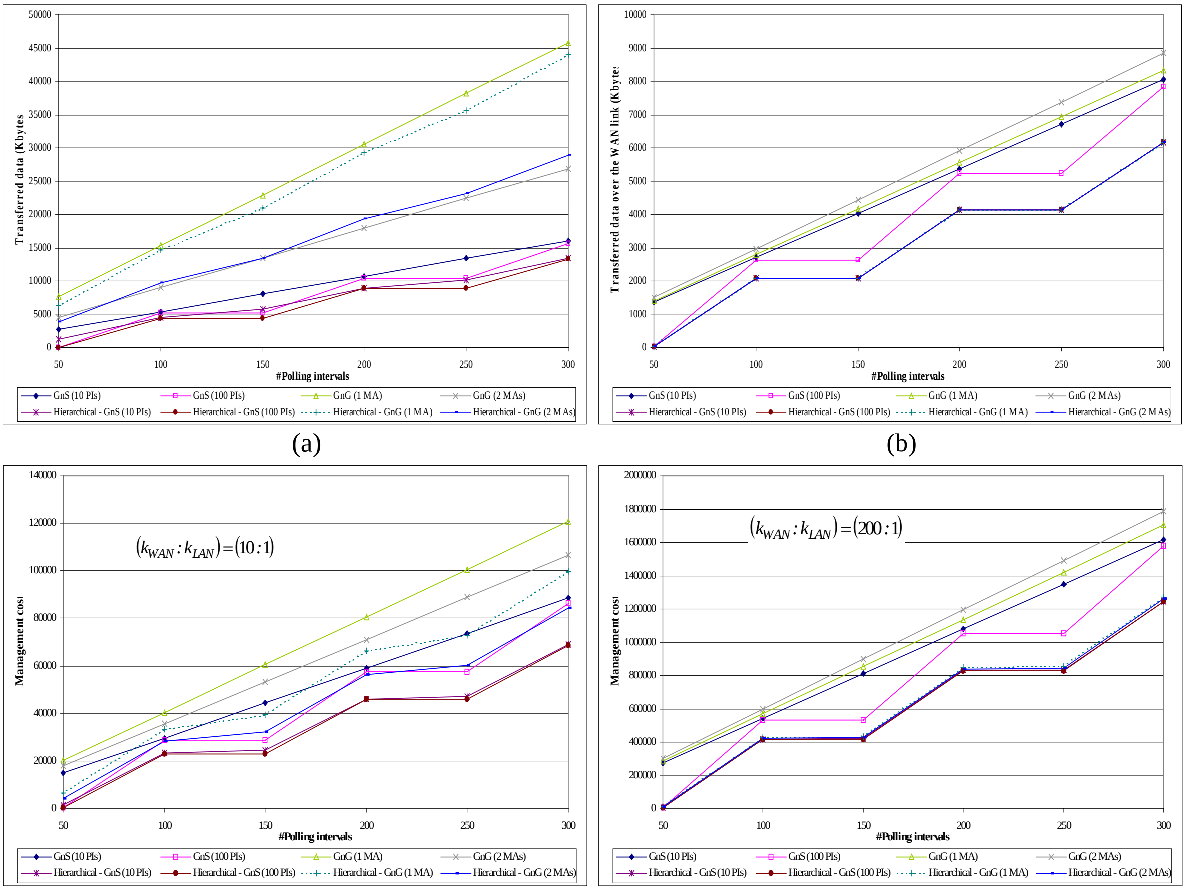
(d)

Figure 6.16. Network overhead measurements for data samples of 50 bytes: (a) Overall management traffic, (b) Network traffic through the WAN link, Overall management cost in the case that the cost coefficient for the WAN link is (c) 10, or (d) 200 times higher than for the high-speed LAN

The overall network traffic is illustrated in Figure 6.16a and Figure 6.17a, as a function of the number of PIs. As expected, the plain GnG approach scales worse in all cases. The combination of hierarchical model with GnG polling slightly reduces network overhead as it localises MA transfers within the MDM's local domain. Accordingly, the coupling of hierarchical management with GnS polling improves the performance of the plain GnS approach (that has also been expected).

The suitability of our hierarchical model for the management of geographically dispersed networks is depicted in Figure 6.16b and Figure 6.17b, which report the data volumes transferred over the 'WAN link' connecting subnets S_0 and S_1 (see Figure 6.15). Since that link does not physically exist, the curves actually depict the traffic sent/received by the manager host. It is noted that the various flavours of the hierarchical result in transferring identical data volumes, as the traffic over the 'WAN link' is restricted in aggregated data deliveries from the MDM to the manager.

An invited side-effect of the hierarchical approach is that following the MDM deployment, transferred NSM data are associated with the management of only *N*-1 hosts, out of the overall *N* managed devices. That is because the MDM resides on a managed NE, with all the management interactions between the former and the later taking place locally. For that reason, whenever the MDM applies GnG polling, its hosting device is visited *last* by travelling MA objects in order to avoid an unnecessary migration.



(a) (b)

(c) (d)

Figure 6.17. Network overhead measurements for data samples of 2000 bytes: (a) Overall management traffic, (b) Network traffic through the WAN link, Overall management cost in the case that the cost coefficient for the WAN link is (c) 10, or (d) 200 times higher than for the high-speed LAN

The network traffic comparisons discussed so far have not taken into account the increased cost associated with transferring data over a low-bandwidth link rather than over high-speed links. Figure 6.16c,d and Figure 6.17c,d illustrate the overall management cost when the ratio of the WAN link over the LAN link cost coefficients is $\left(k_{S_o,S_1} : k_{S_1,S_1}\right)$=(10:1) or (200:1). It can now be observed that the coupling of the hierarchical model with GnG polling represents a cost-effective approach, as MA transfers are not performed over the 'expensive' link. However, the combination with GnS polling remains the most efficient solution. To further reduce the utilisation of the 'WAN link' and, hence, the overall management cost, MDMs should apply a superjacent level of data aggregation, delivering to the manager only higher-level information.

The results graphically illustrated in Figure 6.16 and Figure 6.17 are also analytically presented in Table 6.1.

# Polling Intervals	GnS (10 PIs)	GnS (100 PIs)	GnG (1 MA)	GnG (2 MAs)	Hierarchical - GnS (10 PIs)	Hierarchical - GnS (100PIs)	Hierarchical - GnG (1 MA)	Hierarchical - GnG (2 MAs)
	Overall network traffic, Kbytes (data sample: 50 bytes)							
50	197	33	748	716	133	59	694	677
100	361	178	1473	1408	262	179	1391	1358
150	524	178	2198	2101	335	179	2034	1985
200	688	323	2923	2793	464	300	2732	2666
250	852	323	3647	3485	538	300	3375	3293
300	1016	468	4372	4178	666	420	4073	3974
	Network traffic over the WAN link, Kbytes (data sample: 50 bytes)							
50	115	33	154	253	30	30	30	30
100	197	106	285	483	84	84	84	84
150	279	106	416	713	84	84	84	84
200	361	178	547	943	139	139	139	139
250	442	178	678	1173	139	139	139	139
300	524	250	809	1403	194	194	194	194
	Overall network traffic, Kbytes (data sample: 2000 bytes)							
50	2704	33	7654	4501	1261	59	6347	3842
100	5374	5237	15285	8979	4511	4450	14691	9682
150	8045	5237	22915	13457	5713	4450	20987	13473
200	10715	10442	30546	17936	8963	8840	29331	19313
250	13386	10442	38177	22414	10165	8840	35627	23105
300	16057	15646	45808	26892	13415	13231	43972	28944
	Network traffic over the WAN link, Kbytes (data sample: 2000 bytes)							
50	1368	33	1407	1495	30	30	30	30
100	2704	2635	2791	2966	2078	2078	2078	2078
150	4039	2635	4175	4437	2078	2078	2078	2078
200	5374	5237	5558	5909	4127	4127	4127	4127
250	6710	5237	6942	7380	4127	4127	4127	4127
300	8045	7839	8326	8852	6175	6175	6175	6175

Table 6.1. Comparison of network traffic generated by GnG and GnS polling schemes against hierarchical management approaches

6.6. SUMMARY

This chapter has introduced the concept of *adaptive* hierarchical management that provides a rationale for the use of MA technology. A synopsis of the main contributions of the proposed model follows:

(a) The hierarchical architecture is intrinsically dynamic by employing mobile mid-level managers (MDMs) that may transparently move to a specific network domain to take over its management responsibility and localise the associated traffic. Although hierarchical MA-based management is not an entirely new concept (see [LIO98, SAH98, ZAP99]), our infrastructure goes one step beyond by offering improved adaptability to changing networking environments and defining concrete policies regarding network segmentation into management domains, MDMs deployment, explicit determination of domain boundaries, etc.

(b) Apart form their ability to move from a management domain to another, MDMs may also move within their managed domain. In particular, MDMs periodically inspect the resources availability of their managed nodes and choose to move and resume execution to the least loaded host.

(c) In addition to addressing flexibility issues, management scalability is also further improved by fully exploiting the benefits of agent mobility, as MDMs rely on MAs for the data collection process; these MAs apply filtering operations locally, thereby minimising the volume of data transferred within the individual management domains. The cost associated with the distribution of code implementing new management tasks, is also minimised by applying an efficient "tree multicasting" bytecode distribution technique.

(d) The hierarchical NSM approach implies localisation of network traffic in the individual management domains, thereby providing more balanced traffic distribution and reducing the overhead in the area around the manager station.

(e) Our proposed design ideas are supplemented by a prototype that helped on gaining hands-on experiences and revealing problems related to the use of MAs in distributed management applications.

(f) Fault tolerance issues have been addressed, securing that distributed MDM objects continue to perform their decentralised tasks even if an interconnecting link or the manager platform fails.

(g) An analytical quantitative evaluation oriented to our specific framework design has been undertaken, deriving formulations that define the management cost associated with the proposed architecture.

A prototypical implementation of the introduced MAF has been tested in a realistic topology scenario, comparing its performance against flat MA-based management and plain GnG/GnS polling schemes. The experimental results section, which complements the quantitative evaluation study, has shown that the proposed architecture outperforms the other candidate approaches with sufficient distinct, both in terms of the overall management cost and the bandwidth usage of low-bandwidth WAN links.

CHAPTER 7

NETWORK MONITORING AND PERFORMANCE MANAGEMENT APPLICATIONS

7.1. INTRODUCTION

Research activities in the field of network management incorporating Mobile Agent (MA) technology have attracted much attention, promising to overcome the flexibility and scalability limitations inherent in centralised archetypes. As a result, several Mobile Agents Platforms (MAP) have been proposed for distributed management (see Section 3.6.1), covering a wide spectrum of applications, including traffic analysis [FER01], network testing [GRI97], fault [SUG99, ELD99], configuration [PAG98, KIM98, YUC00] and security [KAR98a, LAZ97] management areas. Following that trend, a number of research papers have investigated the use of MAs for performance management and network monitoring [KU97, ZHA98b, SAH98, CHI99, ZAP99, RUB99, BEL99, PIN99, PUL00b, BOH00c]. However, only a few have identified realistic management applications where MAs could be used for filtering management data, thereby improving the performance of centralised models. Most of these applications focus on the computation of aggregated functions, that 'compress' a number of Management Information Base (MIB) objects to a single value [ZAP99, PIN99, PUL00b] or on sending QoS alarms and delivering periodic summarisation reports by observing raw information [BOH00c]. These applications certainly represent an effective use of MAs, nevertheless, there still are several critical limitations/omissions, summarised in the following:

(i) The MAPs used for developing the aforementioned applications are typically heavyweight (see Section 3.6.1.10). That results in prohibitively 'costly' MA sizes that may seriously affect network performance should frequent MA transfers are involved. The impact of MA migrations on management performance has been addressed in [BOH00c], proposing the exclusive use of constrained mobility, i.e. single-hop agents, in distributed management applications.

(ii) No appropriate method for remote processing/filtering of bulk monitoring data stored in Simple Network Management Protocol (SNMP) tables has been proposed so far. That represents a significant problem, considering the continuous growth of these structures (e.g. tables storing IP routing information), which cannot be efficiently retrieved through SNMP, especially when only a small portion of their stored data is needed.

(iii) Network monitoring applications utilising multi-hop MAs have failed to exploit agents *autonomy* and *mobility* features. In other words, their functionality is restricted on performing repetitive tasks and obtaining specified data patterns from a set of managed devices. The collected data are not effectively used by the MA, but are simply carried throughout their operational travel before being delivered to the manager station. The quantitative evaluation of Section 4.5.2 has indicated an exponential growth of management cost with network size, when a constant amount of data is obtained from each visited host. Hence, mobility does not only remain unexploited (apart from potential response time benefits when short-term monitoring tasks are involved), but also contributes in increasing the associated management cost.

The lightweight, dynamic and adaptive management framework introduced in the previous three chapters addresses sufficiently the first problem, enabling the introduction of management operations in a flexible and scalable manner. The results of a number of experiments have been presented, verifying the precedence of our framework over alternative MA-based approaches and its competence against static distributed object technologies. However, these experiments comprised measurements of response time and network overhead when transfers of variable-length *strings* were involved, with the relation to management applications not being evident. It is therefore necessary to assess the suitability of our framework on realistic management scenarios, whilst comparing its performance against that of the dominant management protocol of the TCP/IP world, i.e. SNMP.

In this chapter, we focus on the second and the third of the aforementioned problems introducing Network & Systems Management (NSM) data intelligent filtering applications. These applications enable the ability of MAs for performing decentralised management operations to apply remote processing of bulk monitoring data, whilst exploiting their mobility feature and the knowledge (data) acquired throughout their travel to increase their selectivity on the data obtained in their next visits.

In particular, we describe ways to: (i) aggregate several MIB values into more meaningful performance indicators, (ii) efficiently acquire atomic snapshots of SNMP tables, and (iii) filter tables' contents by applying arbitrarily complex filtering expressions. Regarding the table filtering application, we introduce the idea of *domain* or *global* level filtering. Specifically, we exploit the multi-node movement that MAs often undertake to perform a *superjacent* level (second stage) of data filtering, in domain or even in network level. This is achieved through correlating the results collected by the hosts that have been already visited with the results obtained from the currently visited host.

The ideas discussed in this chapter have been published in the proceedings of the *2000 IEEE/IFIP Network Operations and Management Symposium* (NOMS'2000), with an extended version published in the *Computer Communications* special issue on "Mobile Software Agents for Telecommunication Applications". The first application has been originally introduced in a paper published in the proceedings of the *IEEE International Conference on Communications* (ICC'99). Full references are given in Appendix A.

The remainder of the chapter is organised as follows: Section 7.2 reviews existing approaches on bulk management data retrieval and discusses their limitations. Section 7.3 introduces the first NSM application, i.e. using MAs for computing aggregated MIB object values. Section 7.4 discusses the SNMP table polling application, while Section 7.5 proposes a model for remote filtering of SNMP tables. The results of a number experiments evaluating the performance of the proposed applications are presented in Section 7.6, proving their precedence over traditional centralised NSM in terms of network overhead. Finally, Section 7.7 summarises and draws the conclusions of this chapter.

7.2. BULK MANAGEMENT DATA RETRIEVAL

Since the early days of SNMP, the requirements for managing IP networks have changed considerably. An important change is that the total amount of management information that needs to be transferred has increased greatly [OLI99]. The management operations defined in the IETF approach are usually low-level, as the management station can typically only get and set atomic MIB object values. Semantically rich operations, such as 'get-column' or 'get-table' are not available yet.

Five problems associated with SNMP-based bulk data transfers have been identified and summarised in the following:

(a) *End-to-end latency*: Using SNMP for retrieving large volumes of MIB data involves a high number of Protocol Data Unit (PDU) exchanges over the network. That results in increasing end-to-end latency, mainly due to the asynchronous nature of SNMP operations, i.e. a `get-next` request is typically sent after a `get-response` corresponding to the previous request has been received by the manager. Potential network problems due to the unreliable nature of the transport protocol (UDP), such as dropped PDUs and retransmissions may also contribute to increasing latency. In addition, the response time of SNMP operations is also affected by the length of the management 'control loop' between the manager and the managed devices, i.e. the number of subnets (hops) that individual request/response packets need to traverse.

The latency problem can be partially addressed through implementing the algorithm described in RFC 1187 [ROS90], which suggests the use of multiple threads in parallel where each thread retrieves only a portion of the table. However, this approach does not reduce the number of PDUs exchanged and fails to shorten the 'control loop', while causing potential overload problems at the agent side due to bursty traffic.

(b) *Network Overhead*: SNMP frameworks are inefficient in terms of the bandwidth utilisation when bulk NSM data are transferred over the network. In addition to the large number of PDUs exchanged, the high overhead is also due to the inefficient Basic Encoding Rules (BER) encoding and the Object Identifier (OID) naming scheme, which introduces a high degree of redundancy. These two problems have been discussed in Section 2.4.3.

(c) *SNMP MIB tables retrieval*: SNMPv1 incorporates an inefficient scheme for MIB table transfers. When using the `get-next` operator, the retrieval of large tables with many rows requires at least one `get-next` operation per table row, increasing the total time needed to transfer a set of management data between the agents and the manager. Should a table row cannot be accommodated into a single PDU (due to message size constraints) even more operations per row are needed. Versions 2c & 3 of SNMP provide a more efficient means of transferring bulk management data, through the `get-bulk` operator [STA99]. Still, the message size constraint applies also in this case, resulting in a large number of PDUs exchanged when considering the transfer of management data in the order of MByte. Additional problems related to MIB tables arise when considering tables that have rows in which some columnar objects do not exist; in other words, tables that allow their row entries to have 'holes' in them. This issue is discussed in [SPR99].

(d) *Data inconsistency*: Another problem associated with bulk management data transfers is that the manager cannot guarantee to retrieve SNMP tables in a consistent state [GOL95]. This is particularly true for large tables, as their retrieval involves a large number of PDU exchanges. The non-negligible time intervals between the retrievals of individual table rows (these intervals highly depend on the length of the management 'control loops') lead to potential inconsistencies. Namely, the agent may update the table contents before completing the table retrieval; in such case different sections of the table will reflect updates at different times, resulting in an inconsistent view of the table. In general, race conditions between concurrent updates of MIB data and their remote retrieval may result in management observations missing critical elements of the corresponding sample behaviour sequence.

(e) *Remote filtering*: In a typical management scenario, the manager may need to search for a single value in a table. In SNMP management the whole table has to be transferred from the remote element to the management station, where the table rows are searched for the value [BAL97]. Hence, large tables will impose heavy network traffic load and will result in computational bottlenecks on the management station. The OSI Systems Management (OSI-SM) [ISO91a] addresses some aspects of management decentralisation through scoping/filtering [ISO91b] and also the Metric Monitoring [ISO93] and Summarization [ISO92] systems management functions: the searching logic is executed at the device and only high-level values are transmitted to the manager. The drawback of this approach is that the logic implementing remote filtering operations is 'hard-coded' on the network elements (NE); such functionality needs to be agreed upon and standardised beforehand [BOH00b].

Sprenkels and Martin-Flatin [SPR99] proposed three alternative approaches to solve the problems associated with bulk management data retrieval. In the first, the authors suggest an SNMP-over-TCP scheme to overcome the UDP packet length constraint, replacement of BER encoding, compression of MIB data and introduction of a `get-subtree` operation. The second approach involves the operation of SNMP in conjunction with the File Transfer Protocol (FTP), whereby every SNMP agent implements two specific MIB modules, saves a certain amount of MIB data into a file and uploads that file to a particular FTP server. The third approach goes one step further proposing the replacement of the SNMP itself by another protocol running over TCP (e.g. HTTP). In effect, all these approaches still imply increased dependency in network resources while involving moderate to radical modifications on existing SNMP frameworks, which cannot be easily realised in short-term. As such, they could only be used to solicit discussions on future Internet management frameworks and protocols. In addition, the approaches proposed in [SPR99] do not consider data filtering as a means to reduce the network overhead imposed by bulk management data retrieval.

We believe that the limitations of SNMP highlighted above may be addressed in a more efficient way, whilst protecting companies' investments on existing frameworks and taking advantage of the huge installation basis of SNMP. In particular, using an MA-based approach, sequences of primitive operations can be grouped into higher-level operations, sent to the NEs and executed locally, independently of the manager station. This brings forward the benefit of flexibility and scalability into the management architecture and provides improved performance by reducing the number of messages exchanged between the management agents and the manager platform, thereby limiting the overall network overhead. Latency and data consistency problems are addressed by local management interactions, which considerably shorten the management 'control loop', whilst semantic compression of management data may be applied to NSM data prior to their transfer to the manager station.

In the following sections, we describe three novel applications of MAs on network monitoring, demonstrating their ability to sufficiently address *all* the aforementioned problems of bulk management transfers.

7.3. HEALTH FUNCTIONS EVALUATION

Developing effective technologies to support compression of real-time management information is a crucial problem in NSM. In practice, the majority of Network Management Systems (NMS) are passive and offer little more than interfaces to raw or partly aggregated/correlated NSM data, typically retrieved through centralised polling. Manager platforms often rely on frequently inspecting the values of specific MIB objects to ensure that managed devices operate on a 'healthy' state. Cases often occur however, where one or two MIB variables are not a representative indicator of systems' state and hence an aggregation of multiple variables is required. A method to achieve efficient data aggregation/compression is to use *health functions* (HF), defined in [GOL93a] as linear combinations of several MIB objects that provide single indices of network state.

Observations of operational variables within the SNMP model are typically made through inspecting the values of MIB objects whose data type is *counter* or *gauge* [STA99]. A counter represents a cumulative (integral) of an operational variable. Typically, only the change in the counter versus its value provides useful indication of the network state. For example, the MIB-II [McC91] counter `ipOutRequests` accounts for the total number of datagrams sent by a network host since its initialisation. Only the rate at which this counter changes represents an indication of the network state. HFs typically combine MIB variables to more useful status indicators. For instance, *five* MIB-II objects are combined to define the percentage *E(t)* of IP output datagrams discarded over the total number of datagrams sent:

$$E(t) = \frac{(ipOutDiscards + ipOutNoRoutes + ipFragFails) * 100}{ipOutRequests + ipForwDatagrams} \qquad (7\text{-}1)$$

The objects appearing in the fraction's numerator represent the IP output datagrams discarded for various reasons (e.g. no route could be found, fragmentation was not possible, lack of buffer space, etc.), while `ipForwDatagrams` accounts for the number of forwarded datagrams. *E(t)* provides an average sense of discarded datagrams ratio over a time window since boot-time. A useful indication of the instantaneous network state is provided by the derivative $e(t) = E'(t)$. This derivative may be approximated by frequent sampling of the respective managed variables and computations of the *E(t)* value changes over a Polling Interval (PI) equal to *Δt*:

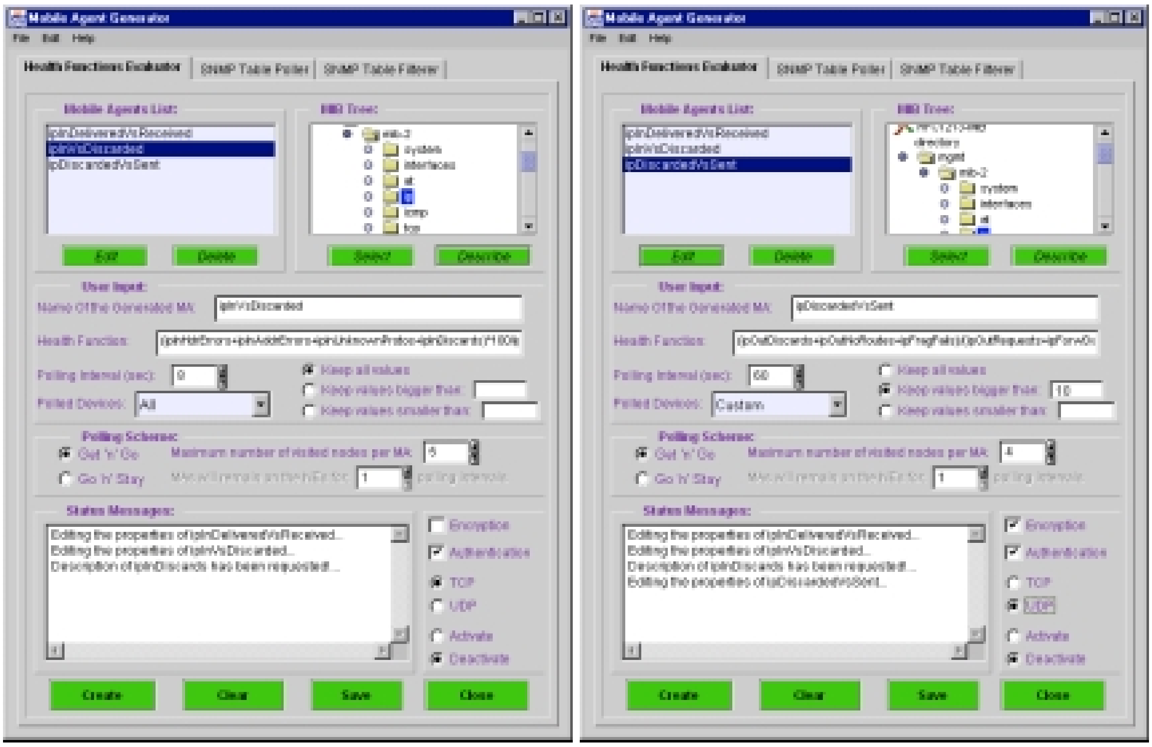
$$e(t) = E'(t) = \frac{E(t + \Delta t) - E(t)}{\Delta t} \tag{7-2}$$

Contemporary NMSs do not support the flexibility and decentralisation required to effectively compute HFs. An HF could not be usefully incorporated as part of a static MIB design, as the specific function used and its parameters depend on the particular installation, configuration and administrative policies. These parameters vary among different systems and during different times within the same network. The OSI-SM supports encapsulation of functions within managed objects and their remote invocation by managing entities [ISO93], yet, these functions are statically bound to a managed object at its design time. Again, designers of managed objects may not provide configuration or time-independent HFs. This problem has been addressed in [PAV6] which proposes extensions to the management functions mentioned above, with the intention to eventually lead to fully dynamically defined monitoring policies without having to re-design the managed objects.

In addition, HFs cannot be usefully computed by centralised managers, as the polling rates required to aggregate the MIB variables may far exceed platforms processing capability; considerable network overhead may also be generated, as a result of failing to perform semantic data compression and reduce data volume at the source. Regarding the HF of Eqn. (7-1), the least 'expensive' option when performing SNMP polling would be to group the five OIDs in a single `get-request` packet. The response packet would then include the OIDs along with the requested values, with the OIDs occupying more space than the actual values. These request/response packets are typically transmitted regardless of the monitored systems health status. A fairly large number of managed devices and frequent polling may result in excessive usage of network resources. Another problem associated with remote polling is that it introduces random perturbations in approximating temporal derivatives of managed variables, leading to errors and potential hazards in management decisions (this is discussed in detail in [GOL93a]).

All these problems suggest that the computation of HFs is more efficiently addressed when performed in proximity to the data source. [GOL93b] examined Management by Delegation (MbD) paradigm as candidate technology to support the flexible and effective evaluation of HFs. Herein, we investigate the potential of MA technology to achieve the same goal. Delegating dynamically customisable HFs to MAs enables direct observation of network behaviours at sufficient precision. HFs may be modified at runtime to reflect varying behaviour patterns at different systems or times. By computing health indicators locally, vast amounts of real-time data can be significantly compressed.

The MAs constructed by the Mobile Agent Generator (MAG) tool (see Section 4.4.1.4) are able to compute HFs, thereby providing a way to semantically compress large amounts of data into single indices representing portions of the system status. Thus, a single value is returned to the manager station, relieving it from processing NSM data, while the MA's state size remains as small as possible. MAs can also be instructed to transmit computed values only in the case that certain thresholds are crossed. Another advantage of relying on MAs to compute HFs is their ability to autonomously execute actions based on the current state of a device. They can take precautionary steps to prevent critical situations, or possibly repair the system without the manager's intervention. Actions for every possible threshold event can be defined in the code of the agent. An action could be triggered if a threshold is exceeded for a certain number of sample intervals so that temporary peaks are filtered out.



Mobile Agents List: ipInDeliveredVsReceived, ipInVsDiscarded, ipDiscardedVsSent

Name Of the Generated MA: ipInVsDiscarded
Health Function: (ipInHdrErrors+ipInAddrErrors+ipInUnknownProtos+ipInDiscards)*100/
Polling Interval (sec): 9
Polled Devices: All
● Keep all values
○ Keep values bigger than:
○ Keep values smaller than:
● Get 'n' Go Maximum number of visited nodes per MA: 5
○ Go 'n' Stay MA will remain in each HE for: 1 polling intervals

Editing the properties of ipInDeliveredVsReceived...
Editing the properties of ipInVsDiscarded...
Description of ipInDiscards has been requested!...

□ Encryption
☑ Authentication
● TCP
○ UDP
○ Activate
● Deactivate

Name Of the Generated MA: ipDiscardedVsSent
Health Function: (ipOutDiscards+ipOutNoRoutes+ipFragFails)/(ipOutRequests+ipForwD
Polling Interval (sec): 60
Polled Devices: Custom
○ Keep all values
● Keep values bigger than: 10
○ Keep values smaller than:
● Get 'n' Go Maximum number of visited nodes per MA: 4
○ Go 'n' Stay MA will remain in each HE for: 1 polling intervals

Editing the properties of ipInDeliveredVsReceived...
Editing the properties of ipInVsDiscarded...
Description of ipInDiscards has been requested!...
Editing the properties of ipDiscardedVsSent...

☑ Encryption
☑ Authentication
○ TCP
● UDP
○ Activate
● Deactivate

Figure 7.1. Setting the properties of two HF evaluators through the MAG GUI

It is noted that either Get 'n' Go (GnG) or Go 'n' Stay (GnS) polling schemes may be used, when monitoring tasks that involve HF computations are considered, depending on the type and execution period of the task. When collected values are intended for off-line analysis, GnS may be employed and multiple samples be obtained before delivered to the manager application, reducing the associated network overhead to a great extend. A downside of using GnG is that the calculation of HF derivatives (see Eqn. (7-2)) takes place at the manager host. Alternatively, the HF values obtained in the previous PI from every visited host would have to be carried within the MAs' state, resulting in larger state sizes and increased network overhead.

On the other hand, when using GnS, derivatives are calculated locally, relaxing the manager station from this task.

Furthermore, when utilising the GnS scheme, the polling frequency can be considerably increased at the expense of a slight increment on network load. That overcomes the limitation of SNMP-based management applications, characterised by a linear relationship between the desired accuracy of monitoring (polling frequency) and the generated network overhead. Through performing local interactions and filtering the data samples which are of no interest, MAs can apply higher polling frequencies without putting any additional burden on the network. HF values returned to the manager can be plotted in graphs updated in real-time (see Figure 5.6).

Definitions (classes) of MAs able to calculate HFs, extend the `MCode.HF` class, which defines the basic properties of an ‘HF evaluator’ agent. This process is fully automated through using the MAG tool, which allows specifying the HF formulation and also a number of additional parameters that determine the polling task operation. In particular, the administrator determines the set of managed devices to be monitored, the polling frequency, the transport protocol used for MA transfers (TCP or UDP), whether obtained values should be encrypted or not, the polling scheme (GnG or GnS), whether MAs return every computed HF value or just the ones exceeding a specified threshold, etc (see Figure 7.1).

Surprisingly, the idea of exploiting code mobility to remotely calculate HFs has been also conceived by fellow researchers and discussed in research papers published nearly at the same time that it was first introduced by the author (June 1999). Specifically, Zapf et al. [ZAP99] examined the evaluation of HFs as a case study for AMETAS MA platform in May 1999. The same idea has been also proposed a year later in [PUL00b].

7.4. SNMP TABLE POLLING

As discussed in Section 7.2, some of the major drawbacks of SNMP are related to the bulk transfer of data and in particular the retrieval of large SNMP tables. The widely deployed SNMPv1 has not been designed for transferring large amounts of data. In addition, the total amount of management information that needs to be transferred has been increased greatly, e.g. IP routing tables and TCP connection tables are continuously growing. Later protocol versions (v2c & v3), apart from their limited installation basis on managed devices, do not answer this problem sufficiently, even though they provide the `get-bulk` operator.

Let us consider the retrieval of an SNMP table consisted of thousands entries. When using the *get-next* operator (SNMPv1 model) the table retrieval requires at least one get-next operation per table row (see Figure 7.2a). Should the number of columns per row is such as to exceed the packet size limit, more than one pair of request/response packets will be required per table row. Network overhead is also affected by the inefficient OID naming scheme, which results in very high information redundancy when obtaining SNMP table snapshots. Apart from the apparent impact on network resources, table polling operations typically experience significant latency and lead to potential data inconsistencies. All these issues have been discussed in detail in Section 7.2.

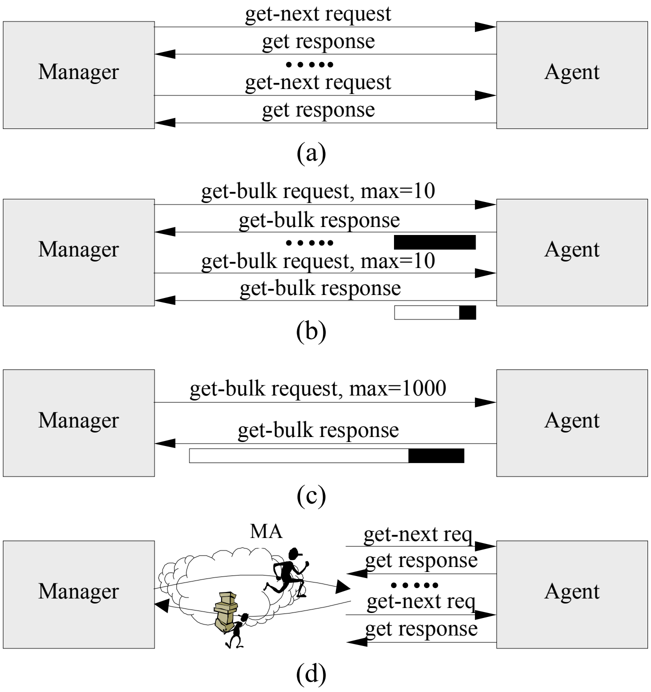
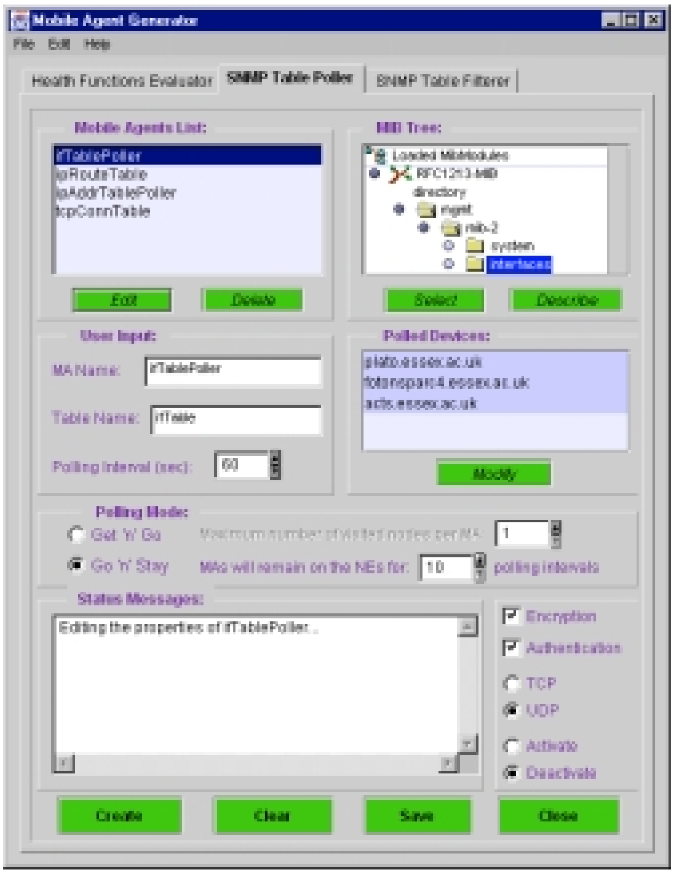


Figure 7.2. Acquiring an SNMP table snapshot through: (a) successive `get-next` requests, (b) multiple `get-bulk` requests, (c) a single `get-bulk` request, (d) MA migration and locally issued `get-next` requests.

The situation improves with the introduction of the `get-bulk` request, which adds to the ability of SNMP to efficiently retrieve large blocks of data by specifying a maximum number of successive values to be returned (*max-repetitions*) [STA99]. That means that the human manager has to guess a value for the max-repetitions parameter. Using small numbers for max-repetitions may result in too many message exchanges (Figure 7.2b). Using large numbers, however, may result in an 'overshoot' effect [SPR99]: the agent returns data that do not belong to the table; these data will be sent over the network just to be discarded (Figure 7.2c). The impact of the 'overshoot' effect is only restrained by the maximum SNMP packet size limit, which also applies to the `get-bulk` PDUs.

Herein, we propose a way to improve the retrieval of SNMP tables both in terms of network overhead and latency. An MA object is dispatched by the manager and visits a pre-determined number of hosts. At each place of contact, when received by the local Mobile Agent Server (MAS), the MA acquires an SNMP table through issuing successive `get-next` requests (see Figure 7.2d). The table contents are then encrypted, if requested, and encapsulated into its state before moving to the next host or returning to the manager. The MA may also obtain several snapshots of the table (with a pre-determined frequency) and wrap them all into its state before delivering to the manager for further analysis (GnS polling scheme). The administrator should be very careful concerning the selection of the appropriate polling scheme. The decision should mainly depend on the size of the SNMP table; moderate or large table sizes may result in dramatic increments of multi-hop MA sizes, causing excessive usage of network resources, pointing GnS as the most appropriate polling scheme choice for this application. This issue is discussed in Section 7.6.1.2.



Mobile Agents List:
- ifTablePoller
- ipRouteTable
- ipAddrTablePoller
- tcpConnTable

MIB Tree:
- Loaded MibModules
  - RFC1213-MIB
    - directory
    - mgmt
      - mib-2
        - system
        - interfaces

User Input:

MA Name: ifTablePoller

Table Name: ifTable

Polling Interval (sec): 60

Polled Devices:
- plato.essex.ac.uk
- fotonsparc4.essex.ac.uk
- acts.essex.ac.uk

Polling Mode:

○ Get 'n' Go Maximum number of visited nodes per MA: 1

● Go 'n' Stay MAs will remain on the NEs for: 10 polling intervals

Status Messages:

Editing the properties of ifTablePoller...

☑ Encryption

☑ Authentication

○ TCP

● UDP

○ Activate

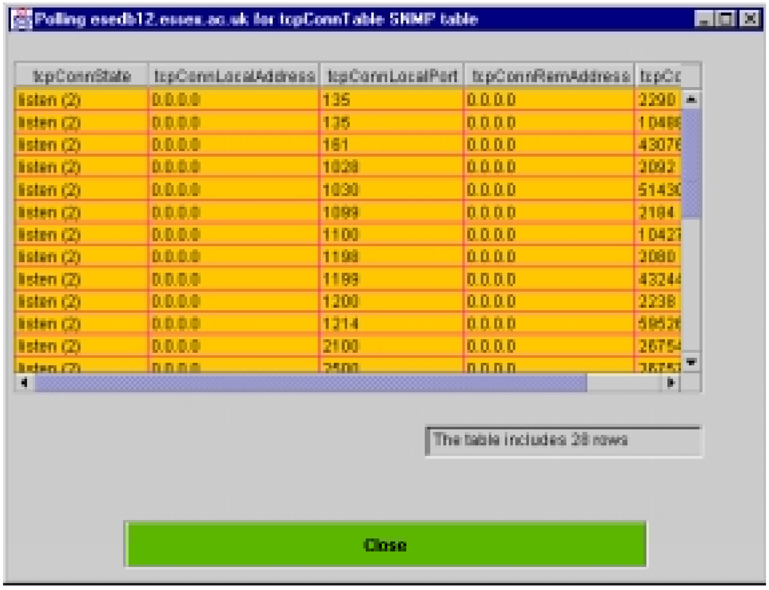
● Deactivate

Figure 7.3. Configuring SNMP table polling operations through the MAG GUI

In addition to limiting the number of message exchanges involved during table transfers, network overhead is further reduced by performing data compression prior to MA migrations (through the Java *gzip* utility). The overall latency is also reduced (especially for large tables), as the round-trip delay of each request/response message exchange (the 'control loop' length) is significantly shorter. An invited side effect of that is the improved consistency of the acquired values. Experimental results on table transfers latency are reported in Section 7.6.1.2.

The SNMP table is encapsulated into the MA's state as a two-dimensioned array. That solves the OID redundancy problem, since the OIDs are not returned to the manager at all; the table's OID (which is the common prefix for all the table objects) added to the value's location into the array (column, row) is sufficient to build the corresponding object's OID (that could also be accomplished by modifying SNMP).

Definitions (classes) of MAs able to obtain table snapshots, extend the `MCode.TP` class, which defines the basic properties of a 'table poller' (TP) agent. New polling operations, used to obtain specific SNMP table views may be added/modified in runtime, specifying the SNMP table and the hosts to be polled, the PI, etc (see Figure 7.3). Table snapshots can be graphically presented to the administrator through the Graphical User Interface (GUI) shown in Figure 7.4.



Polling esedb12.essex.ac.uk for tcpConnTable SNMP table

tcpConnState	tcpConnLocalAddress	tcpConnLocalPort	tcpConnRemAddress	tcpCc
listen (2)	0.0.0.0	135	0.0.0.0	2290
listen (2)	0.0.0.0	135	0.0.0.0	1048
listen (2)	0.0.0.0	161	0.0.0.0	4307
listen (2)	0.0.0.0	1028	0.0.0.0	2092
listen (2)	0.0.0.0	1030	0.0.0.0	5143
listen (2)	0.0.0.0	1099	0.0.0.0	2184
listen (2)	0.0.0.0	1100	0.0.0.0	1042
listen (2)	0.0.0.0	1198	0.0.0.0	2080
listen (2)	0.0.0.0	1199	0.0.0.0	4324
listen (2)	0.0.0.0	1200	0.0.0.0	2238
listen (2)	0.0.0.0	1214	0.0.0.0	5952
listen (2)	0.0.0.0	2100	0.0.0.0	2675
listen (2)	0.0.0.0	2500	0.0.0.0	2675

The table includes 28 rows

Figure 7.4. Graphical display of SNMP table contents

Regarding the implementation of the MA-SNMP agent interaction, instead of reinventing the wheel and developing everything from scratch, we re-used publicly available software as much as possible. Thus, we have used several classes of the AdventNet package [AdventNet] abstracting MIB nodes, issuing SNMP requests, receiving SNMP responses, etc. AdventNet classes have been used as the skeleton upon which higher-level management operations have been built. For instance, table snapshots are obtained through simply invoking the `getTable()` method of the `MCode.SFC` class. Currently, only SNMPv1 is supported, mainly due to its simplicity and because this protocol version is the only one implemented by the Windows NT 'SNMP service'. In future extensions we intend to implement the 'get-table' operator through sending successive `get-bulk` requests aiming at further reducing latency and improving data consistency.

Snapshot views of SNMP tables can be very useful in a variety of applications aiming at investigating transient problems of short duration or identifying trends on changing networking conditions. For instance, a management application may need to retrieve the IP routing table of a router in order to analyse routing configuration patterns. Using SNMP, this retrieval is likely to be very slow and some of the variable values may be updated before the retrieval completion. A management application could be misled and deduce wrong conclusions due to such concurrent updates. It is therefore necessary to obtain atomic table snapshots, in which the values of the MIB variables being retrieved are guaranteed to be consistent or at least are less likely to change during the retrieval. The router's IP routing table (MIB-II `ipRouteTable`) views can be timestamped to identify the time the corresponding snapshots have been taken. Intermittent routing conditions can be detected through taking several snapshots of the routing table (i.e. a trace of views) at specified intervals, and later evaluate the differences between them to accurately analyse the behaviour of dynamic route changes.

7.5. SNMP Table Intelligent Filtering

In most existing network monitoring applications, the retrieved bulk data are usually utilised to feed a processing module responsible for extracting results in a more high-level and 'understandable' form. These results may be used later on to aid on capturing utilisation or error rate peaks, indicate failed devices, foresee possible congestion points, plan future network upgrades, etc. In fact, only a small portion of the obtained values is proved useful as, in by far the majority of cases, bandwidth is consumed to learn nothing other than that the network is operating within acceptable parametrical boundary conditions. This is because the processing action, i.e. the filtering of management data takes place on the manager and *not* on the NE side.

Therefore, we propose a third application of MAs on network monitoring exploiting their ability to dynamically upload management logic in order to perform *intelligent filtering of NSM data on selected SNMP tables*. Specifically, this type of MA is able to acquire an SNMP table and subsequently apply a pre-determined filtering pattern to it. Definitions of MAs able to perform table filtering, extend the `MCode.TF` class, which defines the basic properties of a 'table filterer' (TF) agent.

The filtering operators offered in the current prototype are classified in *Arithmetic* (`Max`, `Min`, `Bigger`, `Less`) and *Textual* (`Match`, `Exclude`) with the corresponding methods implemented in the `MCode.TF` class; the method definitions are shown in Figure 7.5. These operators typically take as input the acquired SNMP table and filter it keeping only the rows for which a given element (defined by its column index) meets certain criteria, e.g. is greater than a threshold value or matches a given text string.

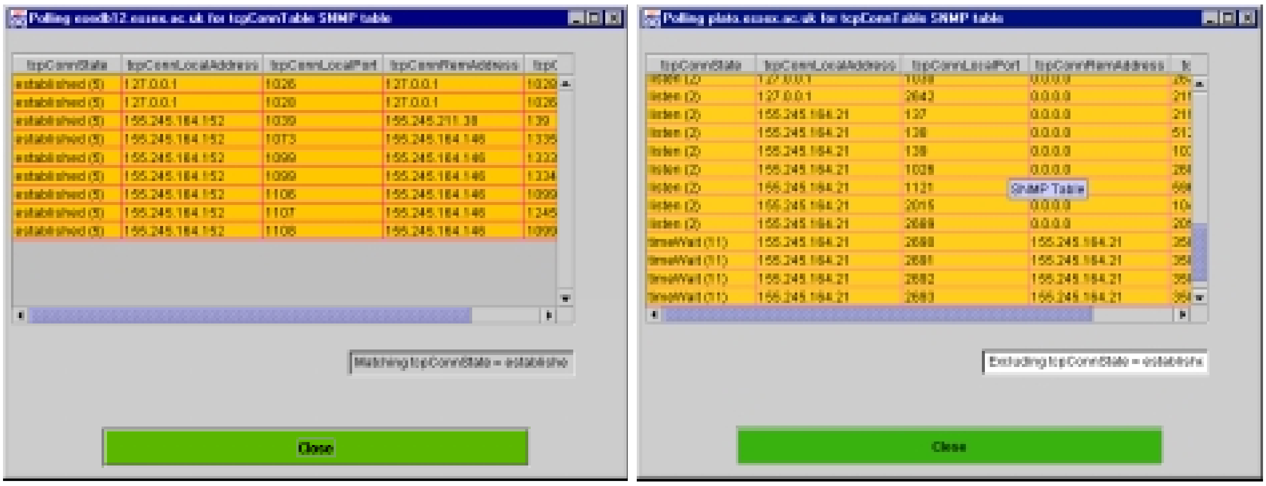
```
String[][] Max (String table[][], int colIndex, int rowsPerHost, int overallRows, boolean
                ascending);
String[][] Min (String table[][], int colIndex, int rowsPerHost, int overallRows, boolean
                ascending);
String[][] Bigger (String table[][], int colIndex, double biggerThan, int rowsPerHost, int
                   overallRows, boolean ascending);
String[][] Less (String table[][], int colIndex, double lessThan, int rowsPerHost, int
                 overallRows, boolean ascending);
String[][] Match (String table[][], int colIndex, String matchedValue);
String[][] Exclude (String table[][], int colIndex, String excludedValue);
```

Figure 7.5. Arithmetic and textual filtering operators method definitions included into the `MCode.TF` class

7.5.1. Textual Operators

When considering textual operators, the TF agent first obtains a snapshot of the requested table (following the procedure described in Section 7.4) and stores it in a two-dimensional array `T[][]` comprising l_T lines. T is passed as an argument to the `Match()` (`Exclude()`) method of the `MCode.TF` class, along with the column index c_i which the filtering operation will be applied upon, and the value (string) v that should be matched (excluded). The method then scans the table and ultimately returns a subset T_F of the original table T, keeping only the table rows that satisfy the condition: $T[j][c_i]=v$ ($T[j][c_i] \neq v$), $j=1 \ldots l_T$. The volume of table values transferred to the manager station may be considerably decreased by performing this simple filtering operation.

An example of using textual operators would be to filter the MIB-II `tcpConnTable` [McC91], which lists information about all the TCP connections of a host. The status of these connections is indicated by the value of the `tcpConnState` table column. Possible connection states are: ‘closed’, ‘listen’, ‘synSent’, ‘synReceived’, ‘established’, ‘finWait1’, ‘finWait2’, ‘closeWait’, ‘lastAck’, ‘closing’, ‘timeWait’ and ‘deleteTCB’. The administrator might be interested in obtaining information referring only to the currently *established* connections. Using SNMP, a management application must retrieve all the entries of the table, and locally select those that match the criteria, i.e. `tcpConnState` = “established”. This retrieval will require many remote network interactions (especially for hosts with hundreds of TCP connections) and therefore will be very slow. Also, the retrieved table row variables may be concurrently updated before the entire retrieval has been completed, resulting in providing inaccurate table views.



Polling essdb12.essex.ac.uk for tcpConnTable SNMP table

tcpConnState	tcpConnLocalAddress	tcpConnLocalPort	tcpConnRemAddress	tcpC
established (5)	127.0.0.1	1026	127.0.0.1	1029
established (5)	127.0.0.1	1029	127.0.0.1	1026
established (5)	155.245.184.152	1039	155.245.211.38	139
established (5)	155.245.184.152	1073	155.245.184.146	1335
established (5)	155.245.184.152	1099	155.245.184.146	1332
established (5)	155.245.184.152	1099	155.245.184.146	1334
established (5)	155.245.184.152	1106	155.245.184.146	1099
established (5)	155.245.184.152	1107	155.245.184.146	1245
established (5)	155.245.184.152	1108	155.245.184.146	1099

Matching tcpConnState = establishe

(a)

Polling plato.essex.ac.uk for tcpConnTable SNMP table

tcpConnState	tcpConnLocalAddress	tcpConnLocalPort	tcpConnRemAddress	tc
listen (2)	127.0.0.1	1030	0.0.0.0	25
listen (2)	127.0.0.1	2642	0.0.0.0	21
listen (2)	155.245.184.21	137	0.0.0.0	21
listen (2)	155.245.184.21	138	0.0.0.0	51
listen (2)	155.245.184.21	139	0.0.0.0	10
listen (2)	155.245.184.21	1028	0.0.0.0	26
listen (2)	155.245.184.21	1121	SNMP Table	59
listen (2)	155.245.184.21	2015	0.0.0.0	10
listen (2)	155.245.184.21	2689	0.0.0.0	20
timeWait (11)	155.245.184.21	2680	155.245.184.21	35
timeWait (11)	155.245.184.21	2681	155.245.184.21	35
timeWait (11)	155.245.184.21	2682	155.245.184.21	35
timeWait (11)	155.245.184.21	2683	155.245.184.21	35

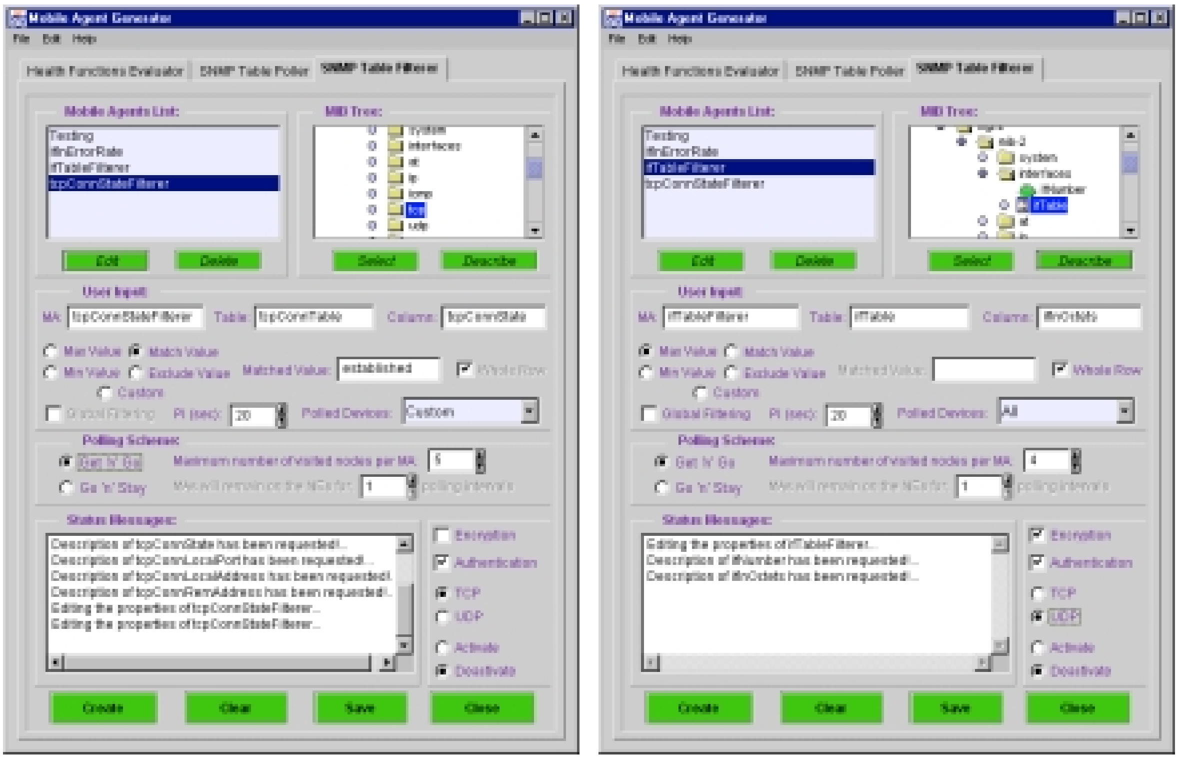
Excluding tcpConnState = establishe

(b)

Figure 7.6. Displaying information returned by a TF agent applying the (a) `Match`, or (b) `Exclude` operator

Using the MA-based approach, the table values related to non-established connections will be discarded by the `Match` operator, preventing unnecessary data transfers (see Figure 7.6a). Likewise, established connections information could be omitted through applying the `Exclude` operator and returning table rows for which: `tcpConnState` ≠ “established” (see Figure 7.6b). The filtering operation could also be based on columns other than `tcpConnState`. For instance, an MA could monitor the number of TCP connections maintained between specific host pairs by checking the `tcpConnRemAddress` column values, or the type of TCP applications running on specific hosts (e.g. `tcpConnRemPort` = ‘21’ would indicate an FTP connection). Filtering patterns making use of textual operators are configured through the MAG GUI (see Figure 7.7a).

In addition to monitoring applications, simple configuration tasks could also be performed ensuring quick response to data updates and high precision. For instance, an MA object could be instructed to selectively terminate TCP connections for specific source/destination hosts. In such case, the MA would first obtain the table rows for which `tcpConnRemAddress` = ‘aaa.bbb.ccc.ddd’ and then modify the operational status for their corresponding connections through a simple `set` request, i.e. `tcpConnState` = “deleteTCB”.



(a)

Mobile Agents List: Testing, ifInErrorRate, ifTableFilterer, tcpConnStateFilterer (selected)

MIB Tree: system, interfaces, at, ip, icmp, tcp (selected), udp

MA: tcpConnStateFilterer Table: tcpConnTable Column: tcpConnState

○ Max Value ● Match Value
○ Min Value ○ Exclude Value Matched Value: established ☑ Whole Row
○ Custom
□ Global Filtering PI (sec): 20 Polled Devices: Custom

● Get 'n' Go Maximum number of visited nodes per MA: 5
○ Go 'n' Stay MA will remain on the NE for: 1 polling intervals

Status Messages:

```
Description of tcpConnState has been requested!...
Description of tcpConnLocalPort has been requested!...
Description of tcpConnLocalAddress has been requested!...
Description of tcpConnRemAddress has been requested!...
Editing the properties of tcpConnStateFilterer...
Editing the properties of tcpConnStateFilterer...
```

□ Encryption
☑ Authentication
● TCP
○ UDP
○ Activate
● Deactivate

(b)

Mobile Agents List: Testing, ifInErrorRate, ifTableFilterer (selected), tcpConnStateFilterer

MIB Tree: mib-2, system, interfaces, ifNumber, ifTable (selected), at, ip

MA: ifTableFilterer Table: ifTable Column: ifInOctets

● Max Value ○ Match Value
○ Min Value ○ Exclude Value Matched Value: ☑ Whole Row
○ Custom
□ Global Filtering PI (sec): 20 Polled Devices: All

● Get 'n' Go Maximum number of visited nodes per MA: 4
○ Go 'n' Stay MA will remain on the NE for: 1 polling intervals

Status Messages:

```
Editing the properties of ifTableFilterer...
Description of ifNumber has been requested!...
Description of ifInOctets has been requested!...
```

☑ Encryption
☑ Authentication
○ TCP
● UDP
○ Activate
● Deactivate

Figure 7.7. Configuring SNMP table filtering operations through the MAG GUI for (a) Textual and (b) Arithmetic operators

7.5.2. Arithmetic Operators

The filtering operation is more complex when arithmetic operators are considered: the user may specify the maximum number of rows that may be returned from *individual* hosts, the maximum number of rows returned from *all* the polled devices, whether the results will be sorted in ascending or descending order, etc.

In that case, filtering may be based on:

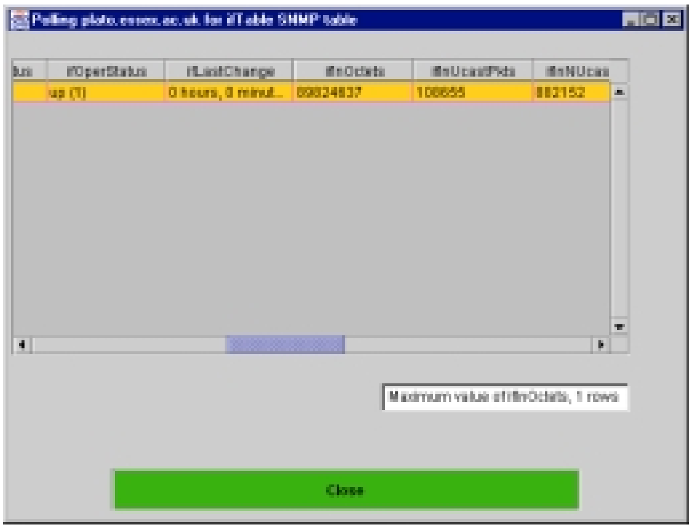
(i) a given table column, e.g. for the MIB-II interfaces table (`ifTable`) [McC91], "*get the row (interface) with the maximum number of received bytes (max `ifInOctets`)*";

(ii) a pre-defined HF, e.g. "*return the two most heavily loaded interfaces for which the utilisation level is higher than 60%*".

The first of the aforementioned examples is implemented through invoking the *Max* method with the following parameters:

```
String[][] maximum = Max (table, 10, 1, -1, true);
```

The arguments passed to the method are: a snapshot of the interfaces table, the column index of `ifInOctets` within `ifTable` (10), the number of table rows acquired from each host (1), an indication that there is no limit on the number of rows returned by all managed hosts (-1) and also request for sorting table rows in ascending order of `ifInOctets` (true). It is noted that TF agents do not necessarily encapsulate the entire table, but may only store (if requested) only selected table columns. Stored results are optionally encrypted prior to their transfer to the manager station. The results returned by the TF agents are displayed on a GUI, shown in Figure 7.8.



Polling plato.essex.ac.uk for ifTable SNMP table

tus	ifOperStatus	ifLastChange	ifInOctets	ifInUcastPkts	ifInNUcas
	up (1)	0 hours, 0 minut...	89824837	108655	882152

Maximum value of ifInOctets, 1 rows

Figure 7.8. Results of `ifTable` filtering

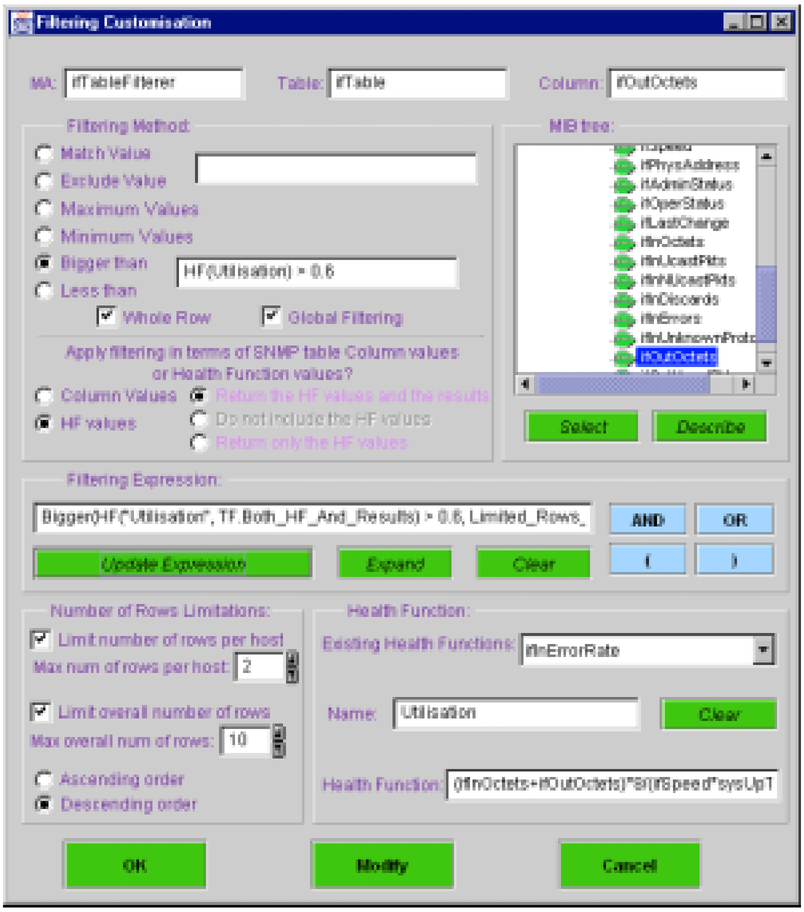
The second example is similarly implemented, through an invocation of the *Bigger* method:

```
String[][] bigger = Bigger (Utilisation (getTable (“ifTable”)), HFcolumn, 0.6, 2, 10, false);
```

which returns the two most heavily loaded interfaces of each host, in descending order, given that their utilisation is greater than 60%. A maximum of ten interfaces will be returned to the manager coming from any of the polled NEs. Should a large number of interfaces are utilised over 60%, only the ten interfaces with the highest utilisation will be returned (the information referring to the rest will be discarded). The *Utilisation* method, defined in the MA’s implementation code, takes as a parameter the interfaces table and calculates the utilisation HF for each one of its rows:

$$U(t) = \frac{(ifInOctets + ifOutOctets) * 8}{(ifSpeed * SysUpTime * 100)} \tag{7-3}$$

The HF values are appended to the interfaces table, which is subsequently sorted into HF values order (using the `Utilities.QuickSort` class). The resulting table is scanned and the rows (two, at maximum) with HF values greater than 0.6 are returned, in descending order. In the case where only specific table columns are requested (e.g. only the number of bytes sent by the most heavily loaded interfaces), the rest of the columns will be discarded. For devices with large number of interfaces (e.g. routers), that filtering method results in significant network load savings. An advantage of using HFs in filtering expressions is that complex HFs incorporating a large number of MIB objects can be defined only *once* and then re-used by other filtering operations. A rich library of HFs can easily be created that way and efficiently utilised to accelerate the customisation of filtering patterns.



Filtering Customisation

MA: ifTableFilterer
Table: ifTable
Column: ifOutOctets

Filtering Method:
○ Match Value
○ Exclude Value
○ Maximum Values
○ Minimum Values
● Bigger than: HF(Utilisation) > 0.6
○ Less than
☑ Whole Row
☑ Global Filtering

Apply filtering in terms of SNMP table Column values or Health Function values?
○ Column Values
● HF values
● Return the HF values and the results
○ Do not include the HF values
○ Return only the HF values

Filtering Expression: Bigger(HF("Utilisation", TF.Both_HF_And_Results) > 0.6, Limited_Rows_

Number of Rows Limitations:
☑ Limit number of rows per host
Max num of rows per host: 2
☑ Limit overall number of rows
Max overall num of rows: 10
○ Ascending order
● Descending order

Health Function:
Existing Health Functions: ifInErrorRate
Name: Utilisation
Health Function: ((ifInOctets+ifOutOctets)*8/(ifSpeed*sysUpT

Figure 7.9. Customising the SNMP table filtering operation parameters

The range of table filtering applications is practically unlimited. Consider, for example, a typical routing device, whose routing table (MIB-II `ipRouteTable`) comprises hundreds or even thousands rows. A management application may need to process all the table entries for which the next-hop field (`ipRouteNextHop`) has certain values. For instance, the next-hop field may represent the IP address of a device, which is due for service. This selection criteria may be specified as "*retrieve all the* `ipRouteTable` *rows for which ipRouteNextHop = aaa.bbb.ccc.ddd*". For an SNMP application, this would require first retrieving all the routing table rows to the management station and subsequently performing the data filtering. Yet, only a few of these are of interest to the management application. Thus, using SNMP involves a large overhead of remote interactions. Significant cost savings may be achieved through applying the proposed MA-based approach.

Simple SNMP table filtering applications are customised through the MAG GUI (see Figure 7.7b). When such applications are characterised by increased complexity, their configuration is performed through the GUI shown in Figure 7.9.

7.5.3. Domain or Global Level Filtering

The filtering applications described in the preceding sections involve obtaining high-level management data from individual devices. Device-oriented statistics are not directly useful to the network administrator should generic views of the entire managed network status are required. Such views can be obtained by *correlating* the data returned by individual hosts; data correlation is typically performed by the manager. In addition to causing processing bottlenecks at the manager platform, this scheme may also generate considerable network load, should the compression ratio achieved over the originally obtained NSM data is not very high.

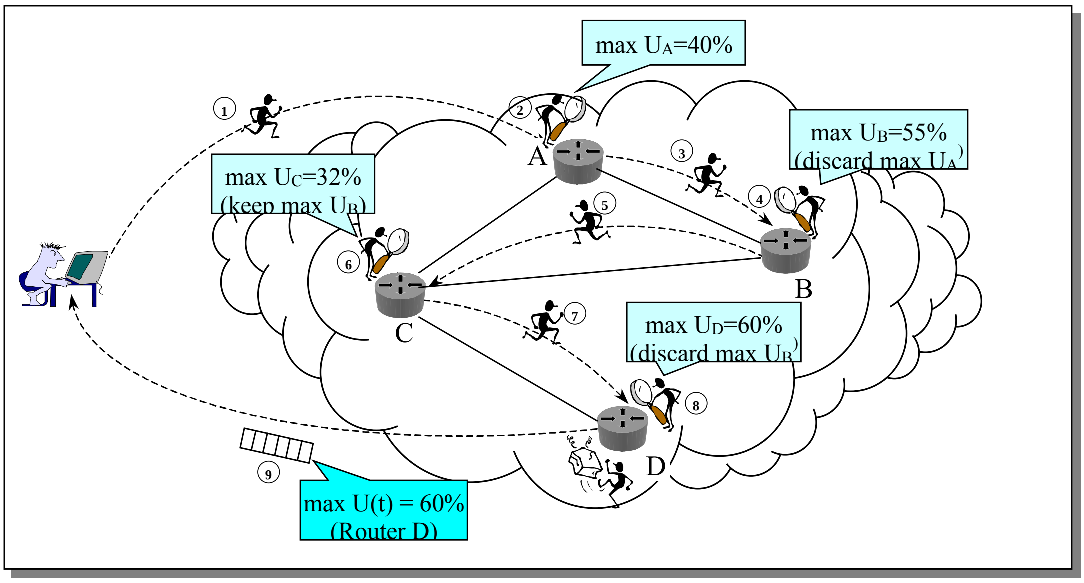
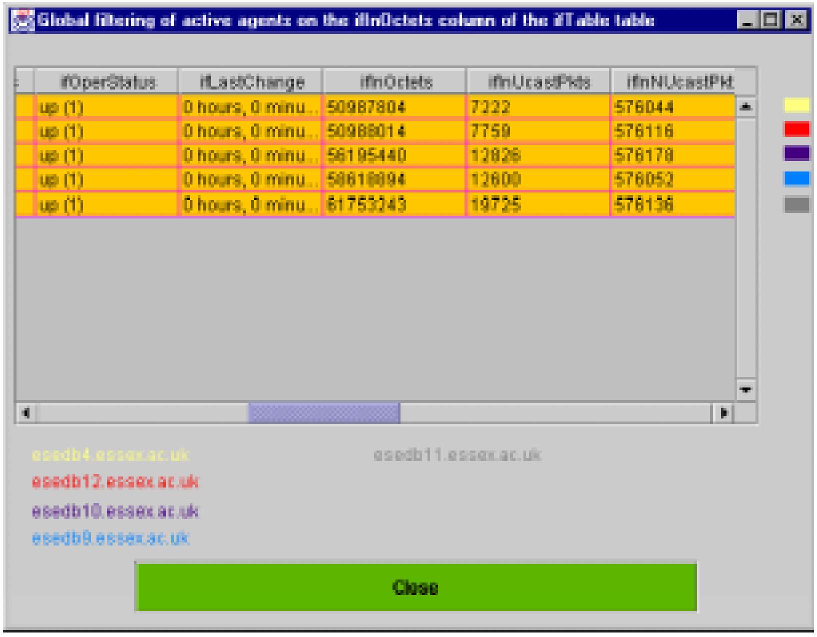

max U_A=40%
max U_B=55% (discard max U_A)
max U_C=32% (keep max U_B)
max U_D=60% (discard max U_B)
max U(t) = 60% (Router D)
A
B
C
D
1
2
3
4
5
6
7
8
9

Figure 7.10. Global filtering applied on the network interfaces of a set of routers

Herein, we propose a more effective and scalable method for obtaining network-oriented statistics through introducing the concept of *domain* or *global level* filtering. In particular, we exploit the multi-node movement of MAs to perform an additional level (second stage) of data filtering, in domain or even in network level. This is achieved by comparing/merging the results already collected with those that have been just obtained/processed. Hence, not only data processing is evenly distributed among network devices, but the MA's state size is prevented from growing rapidly (and therefore the network overhead is further reduced), since the amount of information stored in the MA's data folder practically remains constant.

It should be emphasised that global filtering is well suited for both GnG and GnS polling schemes. In the former case the MA may, for example, return the two most heavily loaded interfaces found in the *entire network*, whereas in the latter, record a utilisation or error rate peak on a host within a given observation period. When employing GnS polling, higher sample rates may be used without putting any additional burden on the network, since this will not affect the MAs state size. In the GnS polling approach, MAs simply correlate 'old' with 'new' data and cannot provide a global view of the managed network since their 'visibility' is restricted on a single device. Global views can only be obtained through multi-hop MAs. Hence, the use of the term "global filtering" in the context of GnS polling, is somewhat unidiomatic.

Figure 7.10 illustrates a management scenario where an MA is assigned the task to return information about the most heavily loaded interface on a network of routers. The MA visits the routers sequentially and computes the usage of their network interfaces. It then compares the value of the most heavily loaded interface found so far against the highest value found on the local router. If the former is higher, no information is collected and the MA migrates to the next router; otherwise, it discards its encapsulated data and stores the MIB values referring to the most heavily loaded interface of the local router. The same process is repeated at every visited router. At the end of its route, the MA will send to the manager the *global* maximum interface utilisation and its corresponding router address.



Global filtering of active agents on the ifInOctets column of the ifTable table

ifOperStatus	ifLastChange	ifInOctets	ifInUcastPkts	ifInNUcastPkt
up (1)	0 hours, 0 minu...	50987804	7322	576044
up (1)	0 hours, 0 minu...	50988014	7759	576116
up (1)	0 hours, 0 minu...	56195440	12826	576178
up (1)	0 hours, 0 minu...	58618894	12600	576052
up (1)	0 hours, 0 minu...	61753243	19725	576136

esedb4.essex.ac.uk
esedb12.essex.ac.uk
esedb10.essex.ac.uk
esedb9.essex.ac.uk
esedb11.essex.ac.uk

Figure 7.11. Results returned by an MA performing global filtering operations over a set of managed devices

Should SNMP was used to accomplish the same task, the manager would first obtain interfaces table snapshots from all routers (through successive `get-next` requests), and then process and correlate the collected data to extract the requested information. A relatively large number of routers, each with several network interfaces, would imply numerous remote interactions and heavy network load. Using a 'constrained mobility'-based approach would significantly improve scalability as distributed single-hop agents would only return to the manager pre-processed data referring to their local device. Yet, data correlation would be performed at the manager platform in order to obtain network-wide generic management views, implying considerable processing burden. Furthermore, when the initial data filtering performed by distributed MAs is not highly selective, large amounts of data may be transferred to the manager station, potentially over low-bandwidth links.

In GnG polling scheme, the results are delivered to the manager and then displayed on a graphical table component, with the interface information drawn in different colours, indicating the host they have been obtained from. Figure 7.11 presents the results returned by an MA which visits a set of 10 devices and returns the table rows corresponding to the *5* most heavily loaded interfaces in this small network within the last observation period.

To epitomise, global filtering reveals a great potential for the use of agent *mobility* for NSM operations, as global statistics cannot be extracted unless the management application has a higher-level view of the managed devices and performs centralised operations upon them. This view has also been expressed in [GOL96]: "*Other computations must be centralised. For example, consider an application that needs to find the least loaded Ethernet segment to install a new device. To make this decision, the application needs to correlate performance data obtained from many segments. Another example is an application that displays an updated network connectivity map. Both examples are centralised because they need to merge and correlate information from several sources*."

We have shown that high-level views of managed elements can be obtained through exploiting the ability of TFs to move from host to host and perform data filtering/correlation in two stages, based on the knowledge/data already collected throughout their travel. Clearly, network overhead savings are greater when MAs encapsulate non-negligible amounts of data, even after processing the originally obtained information. Should data correlation was not used, multi-hop TFs would experience rapid growth of their state size. If, on the other hand, global filtering is applied, TFs state size remains constant as either the information already carried by the MA or the data obtained at the currently visited device is discarded.

7.5.4. Boolean Filtering Operations

In addition to the scenarios already examined, the construction of filtering expressions with increased complexity has also been considered. Thus, MAs may be constructed with the ability to apply arbitrarily complex *boolean* expressions, namely logical *AND* and *OR* operators, correlating individual filtering functions.

An example employing the *AND* operator would be: "*return the interfaces with utilisation 0.6<U(t)<0.8*". The output array *bigger* of the *Bigger* method invocation (mentioned in Section 7.5.2) would then be passed as a parameter to the *Less* operator to apply a second level of filtering:

```
String[][] final_result = Less (bigger, HFcolumn, 0.8, 2, 10, false);
```

In contrast, when the *OR* operator is considered, the two output tables (arrays) resulting from the individual expressions are simply concatenated. For example, to view the TCP connections with either '*listen' or* '*established'* state, the following expression needs to be applied:

String [][] result = *concat* (*Match* (getTable ("tcpConnTable"), tcpConnState, "listen"), *Match* (getTable ("tcpConnTable"), tcpConnState, "established"));

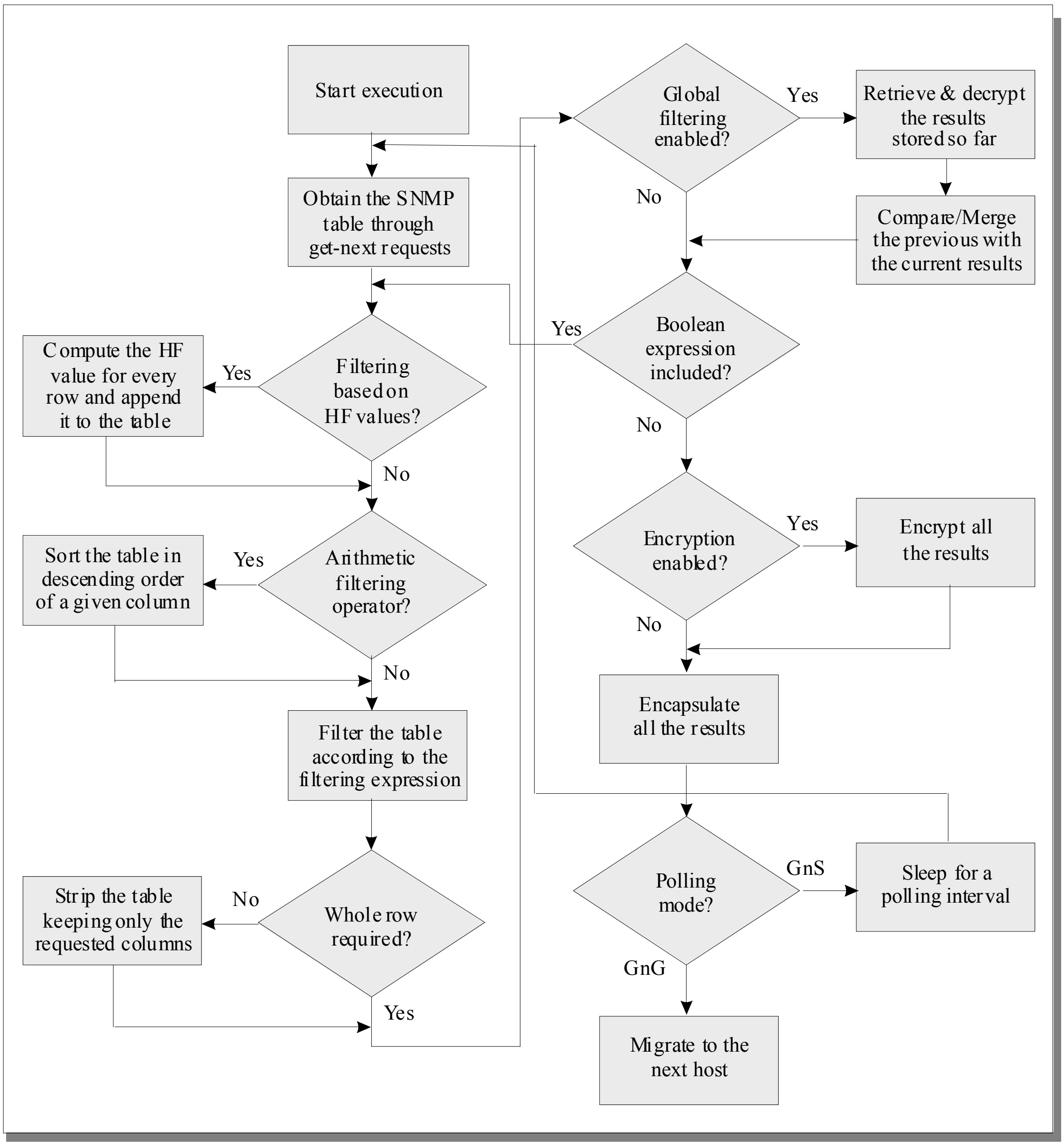
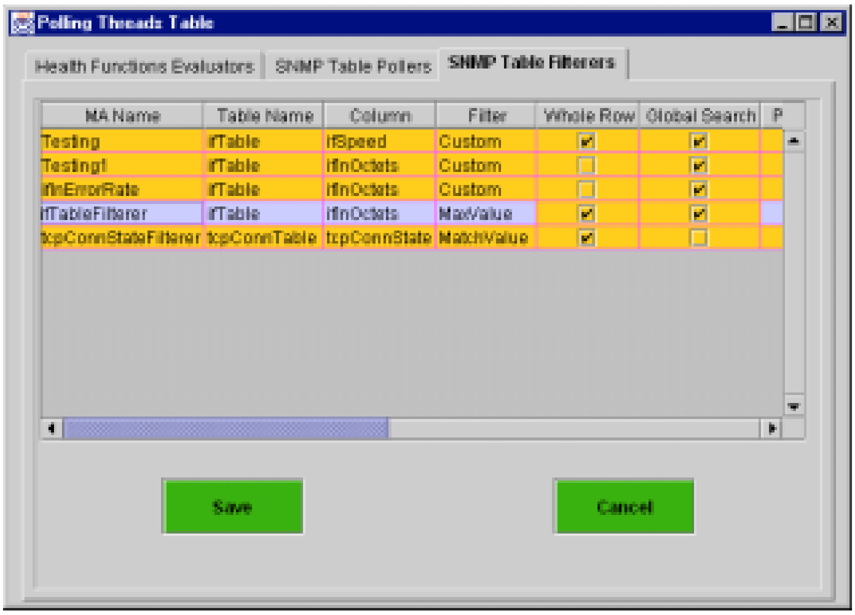

Start execution
Obtain the SNMP table through get-next requests
Filtering based on HF values?
Yes
Compute the HF value for every row and append it to the table
No
Arithmetic filtering operator?
Yes
Sort the table in descending order of a given column
No
Filter the table according to the filtering expression
Whole row required?
No
Strip the table keeping only the requested columns
Yes
Global filtering enabled?
Yes
Retrieve & decrypt the results stored so far
No
Compare/Merge the previous with the current results
Boolean expression included?
Yes
No
Encryption enabled?
Yes
Encrypt all the results
No
Encapsulate all the results
Polling mode?
GnS
Sleep for a polling interval
GnG
Migrate to the next host

Figure 7.12. SNMP table filtering operation flow diagram

The procedure followed by TF agents to filter the contents of SNMP tables, as described in the last sections, is illustrated in the flow diagram of Figure 7.12.

It is noted that following the creation of MAs able to compute HFs, obtain SNMP table snapshots or filter table contents, their corresponding task configuration parameters may be easily modified at runtime through the user-friendly GUI shown in Figure 7.13.



MA Name	Table Name	Column	Filter	Whole Row	Global Search	P
Testing	ifTable	ifSpeed	Custom	☑	☑	
Testing1	ifTable	ifInOctets	Custom	□	☑	
ifInErrorRate	ifTable	ifInOctets	Custom	□	☑	
ifTableFilterer	ifTable	ifInOctets	MaxValue	☑	☑	
tcpConnStateFilterer	tcpConnTable	tcpConnState	MatchValue	☑	□	

Figure 7.13. Runtime configuration of MA-based management applications

7.5.5. Relevant Work

Relevant work has been reported in [GOL96], which describes ways to create MIB views within an agent. Views are created by defining operations on SNMP tables using a View Definition Language (VDL). However, the support of a VDL interface (*VDL translators*) requires extensions to standard SNMP agents and increases the footprint on network devices, while delegation agents need to be always updated as soon as a new control operation is introduced. This is not a prerequisite in the MA-based approach, where the local MAS modules are unaware of the incoming MAs functionality. In addition, the data stored within the created MIB views are queried through standard SNMP interactions, i.e. `get` and `get-next` requests, which may create considerable traffic should the selectivity of the involved management operations is very high.

Anerousis introduced *Marvel* architecture, whose information model supports the generation of computed views of management information [ANE98], thereby increasing the level of semantics available to the management-application designer. Computed views consist of monitoring, control, and event views of management data collected from agents. These data are aggregated into Marvel objects (also known as *aggregated managed objects*) using spatial and temporal filters. Marvel objects may co-reside with management agents or execute at external repositories, depending on the amount of processing required (some aggregations can be very CPU intensive). However, in addition to *statically* define the location of Marvel servers, aggregations of management data are obtained through standard management protocols such as SNMP or CMIP, exhibiting the same scalability problems of the VDL approach.

7.6. EXPERIMENTAL RESULTS

In order to thoroughly evaluate the performance of the proposed management models, we have conducted a number of experiments measuring the response time and the network overhead involved when employing the management applications described in the preceding sections. The performance issues related with the three applications, i.e. computation of HFs, SNMP table polling, and table filtering are investigated separately, with their performance compared against SNMP-based management.

7.6.1. Response Time

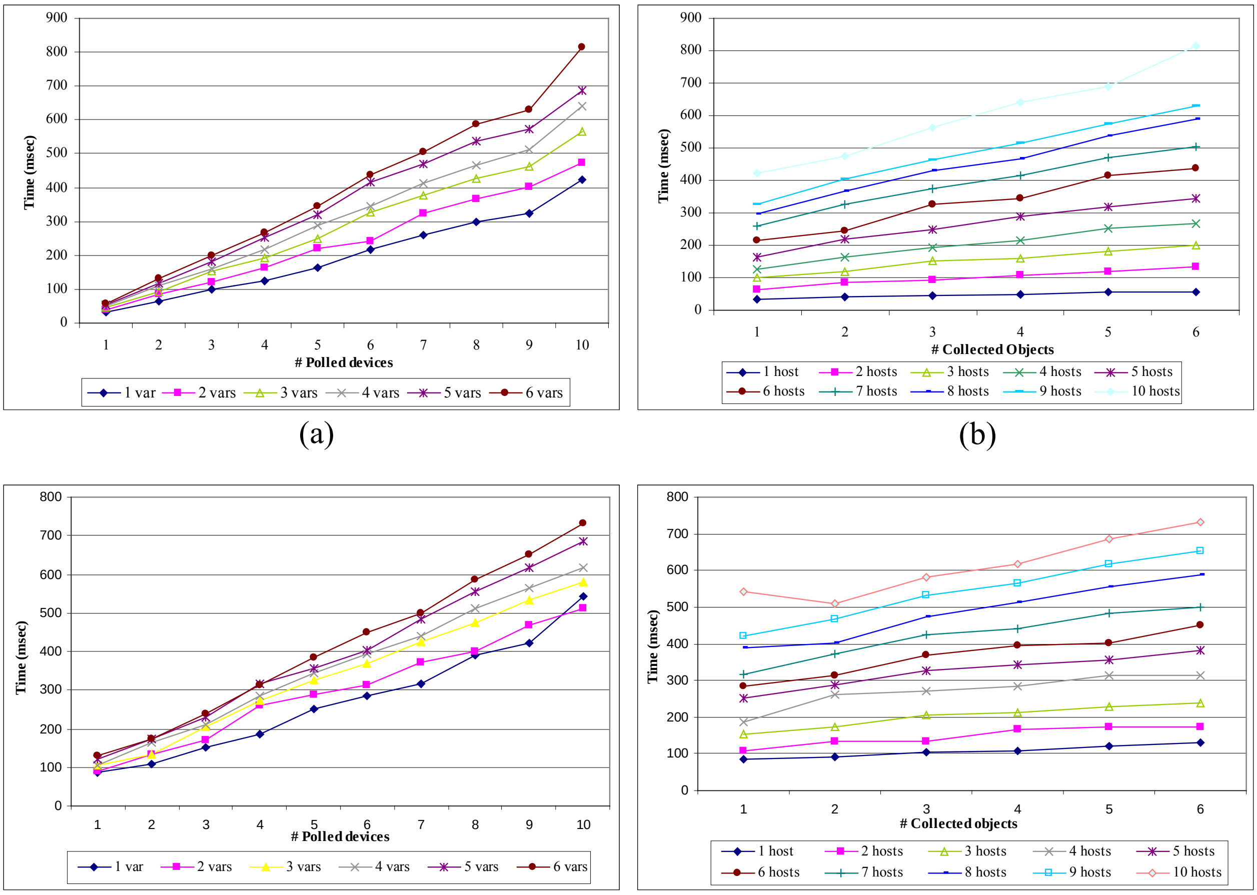
7.6.1.1. Health Functions Evaluation

A. SNMP time measurements

Response time measurements started with an evaluation of the SNMP protocol. The differences between synchronous and asynchronous polling are investigated, while the dependence of the overall time on several parameters, such as the managed network size and the number of collected MIB objects, is also evaluated. Similarly to all previous response time measurements, for every pair of (#polled devices, #MIB objects), experiments have been repeated 100 times with average times and standard deviations subsequently calculated.

As described in Section 4.5.1, in *asynchronous* SNMP polling, `get` requests are sent to all monitored devices with the responses collected in parallel. When a response is received, an event is generated and a callback method automatically invoked, which in turn handles the response packet and extracts the results included within its varbind list. On the other hand, *synchronous* polling involves sequential polling of SNMP agents. The thread that carries out the polling operation remains engaged, waiting the response before sending a `get` request to the next agent. Synchronous/asynchronous SNMP polling operations have been implemented using the AdvnetNet’s Java-based SNMP stack.

The experimental testbed comprises a PC where the manager application executes and a number of PCs varying from one to ten, with active SNMP agents, playing the role of the managed devices. At each polling interval, the manager sends a `get` request to each of the polled devices retrieving the values of specific MIB-II objects (one to six objects), needed to compute the value of an HF. In any case, the retrieval of all MIB objects requires a *single* get/request pair of SNMP packets, as the maximum packet limit is not exceeded.



(a) (b)

(c) (d)

Figure 7.14. Response time as a function of the network size and the collected MIB objects per host for synchronous (a, b) and asynchronous (c, d) SNMP polling operations

The timing measurements are graphically illustrated in Figure 7.14 and indicate a linear increment of response time both as a function of the network size and the number of retrieved MIB objects. Surprisingly, asynchronous polling does not result in a drastic reduction of response time in comparison with its synchronous counterpart. However, the former scales slightly better than the latter (the slope of the corresponding curves are less steep). It should also be noted that the time needed by the manager application to compute the value of the HF for each of the polled devices has not been measured. The results illustrated in Figure 7.14 are detailed in Table 7.1(synchronous polling) and Table 7.2 (asynchronous polling).

# Collected MIB objects

# Polled Devices	1		2		3		4		5		6	
	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
1	32.4	37.3	40.7	38.7	45.4	41.2	48.1	38.1	54.5	43.0	56.8	46.4
2	64.5	43.0	86.8	45.6	93.3	47.0	109.0	49.3	118.4	47.9	133.2	46.2
3	101.3	41.8	119.5	48.0	151.2	49.2	159.2	53.5	180.0	48.0	200.9	50.9
4	125.0	43.6	162.0	47.3	193.4	51.5	215.4	54.0	251.1	57.2	267.6	61.2
5	163.8	44.8	219.8	50.2	248.6	52.4	289.3	59.6	318.4	69.0	344.6	71.1
6	215.7	47.4	243.2	66.9	327.7	115.9	345.6	79.3	415.9	117.5	437.1	108.4
7	259.7	63.1	324.4	71.6	375.7	90.7	413.3	89.1	469.6	117.7	504.6	116.7
8	297.1	81.1	364.9	95.2	428.6	97.3	466.2	104.4	538.0	117.8	588.1	125.6
9	325.3	80.9	403.1	93.1	462.9	91.3	513.0	119.6	573.9	117.9	629.9	144.6
10	422.7	93.1	474.3	98.7	564.7	113.2	641.5	179.3	687.1	148.4	813.1	191.3

Table 7.1. Synchronous SNMP time measurements

# Collected MIB objects

# Polled Devices	1		2		3		4		5		6	
	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
1	85.4	92.0	90.8	85.3	105.6	92.7	106.6	95.2	121.8	102.4	130.7	106.4
2	108.7	78.3	132.3	87.9	133.7	84.6	165.7	114.8	173.3	101.2	174.1	82.3
3	152.2	72.0	172.0	72.5	204.4	74.5	212.3	77.8	228.6	82.3	238.1	89.2
4	187.5	60.3	261.1	99.9	272.6	106.8	283.9	115.7	315.0	115.0	312.6	115.1
5	251.4	88.7	287.1	95.3	325.0	114.5	343.5	100.3	355.2	98.0	383.5	107.3
6	285.1	107.2	313.9	99.0	368.1	95.0	395.2	116.1	403.1	89.3	449.4	113.8
7	315.1	88.6	373.6	111.4	424.6	96.8	440.9	103.9	482.4	109.4	498.9	120.2
8	389.2	117.6	400.4	98.2	475.0	107.6	512.3	115.5	553.7	118.3	586.3	124.0
9	422.2	93.3	468.5	121.5	533.2	102.9	565.8	108.0	618.6	140.7	652.2	132.5
10	541.5	93.7	510.1	125.2	579.8	112.3	616.3	139.0	684.4	159.0	731.7	149.7

Table 7.2. Asynchronous SNMP time measurements

B. Multi-hop MAs time measurements

The response time experiments described in the preceding section have been repeated to measure the time needed to complete the same operations through employing multi-hop MAs rather than SNMP polling. In fact, these experiments do not significantly differ from those presented in Section 4.6.1.2; the only difference is that in this case the obtained management data are real and not ‘simulated’ by encapsulated character strings.

In particular, a single MA object is launched from the manager platform and sequentially visits every managed device. At each point of contact, the MA retrieves a number of MIB object values (varying from one to five), computes and encapsulates the value of an HF and subsequently migrates to the next host included into its itinerary. In addition to the network size and the number of MIB objects retrieved, the effect of the transport protocol used for MA transfers is also evaluated.

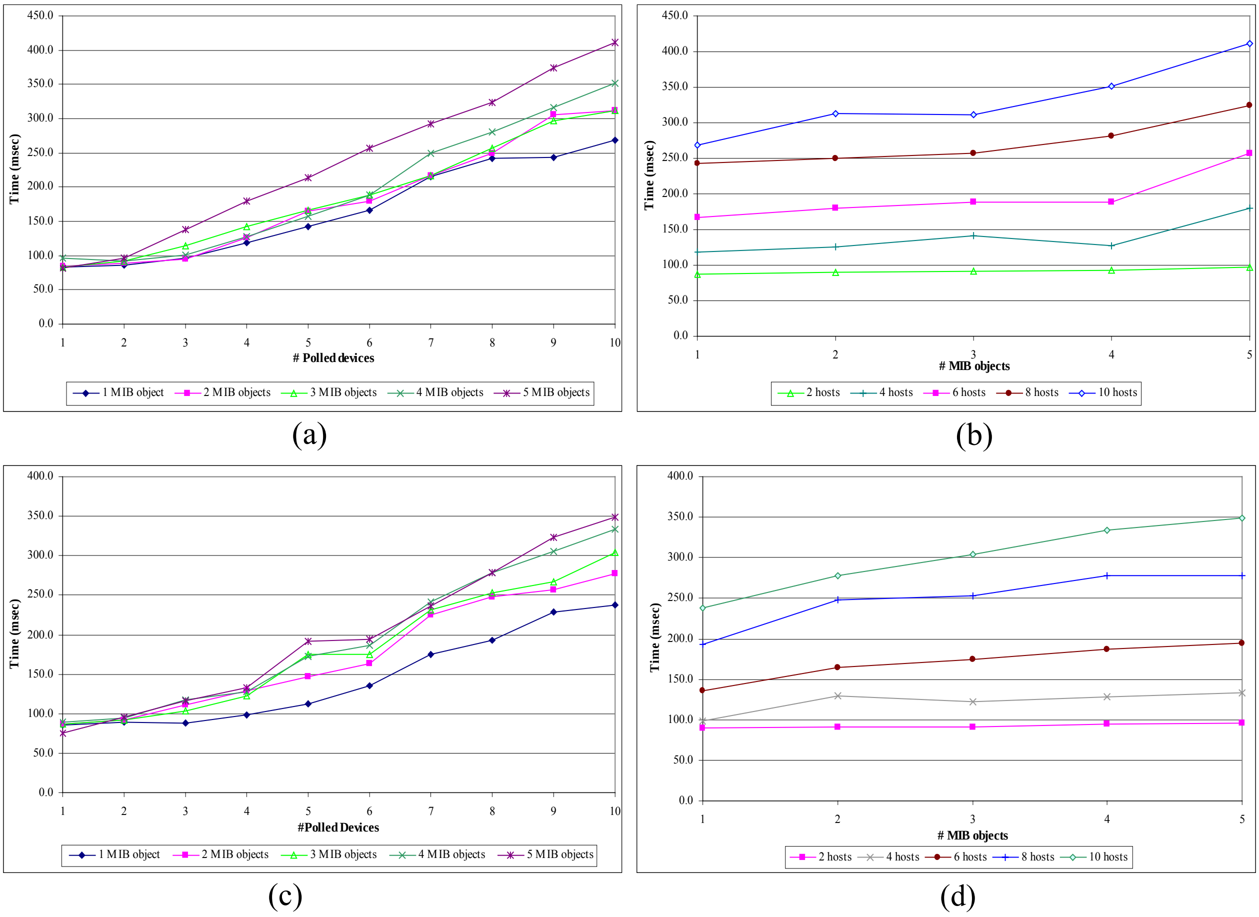


Figure 7.15. Multi-hop MAs response time measurements as a function of the network size and collected MIB objects per host, when the transport protocol is TCP (a, b) or UDP (c, d)

The graphs shown in Figure 7.15 depict an almost linear relationship of response time versus the network size and the number of obtained MIB objects. In addition, the overall latency gain of UDP over TCP-based migrations is marginal, although the former is shown to scale better than the latter as the network size increases. A comparison between Figure 7.15 and Figures 4.25 and 4.26 depicts an increment of the response time of multi-hop MAs for the same number of visited hosts, when real management data are collected. This is due to the time spent on interacting with the SNMP agents and obtain the MIB values. On the other hand, in the experiments described in Section 4.6.1.2, MAs simply encapsulate a pre-constructed string, without any further interaction with the local device.

# Collected MIB objects

# Polled Devices	1		2		3		4		5	
	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
1	82.7	17.7	84.2	21.5	82.5	19.8	96.4	5.9	82.0	6.2
2	86.5	18.9	89.7	22.5	91.6	44.9	91.6	33.0	96.8	18.3
3	97.1	16.7	95.4	15.1	100.9	8.0	114.3	11.6	137.5	17.8
4	118.9	9.3	125.9	14.5	142.0	16.7	127.4	16.6	179.6	24.3
5	142.1	20.6	164.2	19.0	166.2	19.0	157.5	17.9	214.1	18.2
6	166.5	17.6	180.1	19.1	188.9	20.4	188.1	20.0	256.6	16.7
7	215.5	316.0	216.6	13.9	217.1	6.7	249.6	20.7	292.1	22.5
8	242.7	21.8	249.9	23.1	257.5	11.1	281.1	6.6	324.4	15.6
9	242.8	17.5	296.5	16.3	306.0	17.1	316.7	27.1	373.7	20.0
10	268.9	24.6	312.5	16.4	311.8	17.7	352.0	19.0	412.0	29.5

Table 7.3. Response time measurements for multi-hop MAs, when using TCP for MA transfers

# Collected MIB objects

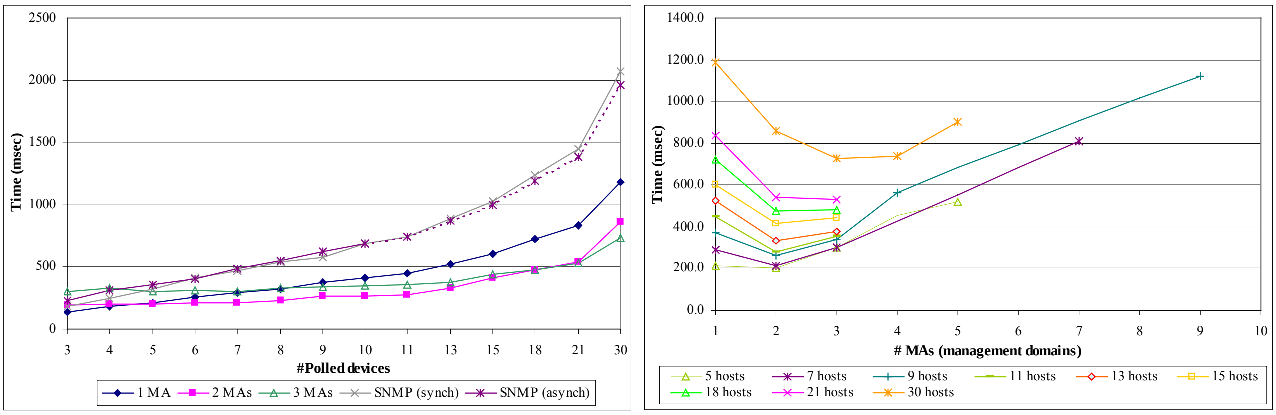
# Polled Devices	1		2		3		4		5	
	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
1	86.0	7.5	86.3	5.2	86.7	13.5	89.8	10.6	76.0	6.2
2	89.2	10.7	91.4	5.4	91.4	8.3	94.5	11.1	95.3	4.1
3	88.8	19.1	110.6	6.7	103.8	9.1	117.4	8.2	116.7	5.9
4	97.9	10.4	129.0	12.4	122.2	10.8	128.2	12.6	132.8	10.9
5	113.0	10.2	147.0	10.9	175.4	19.3	172.6	7.4	191.7	21.0
6	135.3	9.3	164.1	17.3	175.0	11.1	187.1	8.7	193.9	10.8
7	175.1	16.0	224.5	13.7	231.2	11.8	241.4	11.5	236.5	9.1
8	193.0	13.4	247.7	20.9	253.5	11.2	278.0	22.2	278.1	10.9
9	228.6	21.6	256.5	13.8	266.6	19.8	305.9	15.2	323.4	19.3
10	237.5	33.5	277.4	18.7	304.2	30.7	333.7	22.0	349.4	29.0

Table 7.4. Response time measurements for multi-hop MAs, when using UDP for MA transfers

Interestingly, the multi-hop MAs-based polling scheme exhibits reduced latency in comparison with SNMP polling, for the same number of polled devices and retrieved MIB objects. It should be admitted though that SNMP-based operations latency would probably be lower should a non-Java implementation of the SNMP stack was used (due to the performance limitations of Java). The results illustrated in Figure 7.15 are also presented in detail in Table 7.3 (MA transfers over TCP) and Table 7.4 (MA transfers over UDP).

C. GnG-based polling time measurements

A last set of timing experiments related to HFs computation, aims at evaluating potential latency gains when replacing the ‘flat’, multi-hop MAs-based polling by GnG polling.



(a) (b)

Figure 7.16. Response time measurements for various GnG scheme configurations, as a function of (a) the network size, and (b) the number of launched MAs (management domains)

# MAs launched per Polling Interval

# Polled Devices	1		2		3		4		5	
	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
3	137.5	8.6	189.4	9.1	298.6	22.3	-	-	-	-
4	179.6	23.0	198.0	17.9	-	-	406.7	24.4	-	-
5	214.1	15.3	201.6	15.6	300.7	18.0	-	-	522.2	40.0
6	256.6	20.2	209.0	21.0	308.7	18.5	-	-	-	-
7	292.1	34.4	213.1	25.1	301.6	45.4	-	-	-	-
8	324.4	6.9	232.1	22.5	328.7	2162.8	434.8	43.7	-	-
9	373.7	17.4	264.4	18.6	336.4	41.3	561.8	65.0	-	-
10	412.0	7.6	267.4	23.6	349.7	56.7	-	-	575.6	73.8
11	448.3	18.7	276.8	23.8	357.7	33.9	-	-	-	-
13	525.9	12.7	334.1	28.9	378.3	50.0	-	-	-	-
15	603.5	38.5	416.1	37.9	440.7	37.6	-	-	-	-
18	719.8	42.5	474.0	65.5	478.9	46.0	-	-	-	-
21	836.2	50.1	539.6	64.3	532.8	49.9	-	-	-	-
30	1185.3	58.5	860.0	108.6	729.5	73.5	740.4	69.2	902.4	88.4

Table 7.5. Response time measurements for several GnG polling scheme configurations

In particular, the manager platform partitions the managed network into a number of management domains and launches an equal number of MAs per PI, each assigned to a specific domain. At each point of contact, the MA retrieves *five* MIB object values needed to evaluate a HF, computes and encapsulates the HF value and subsequently migrates to the next host included into its itinerary. TCP has been used as a transport protocol for MA transfers. Timing measurements are graphically illustrated in Figure 7.16 and analytically presented in Table 7.5. Figure 7.16a shows that a GnG scheme employing two or three MAs optimises the overall latency for networks comprising more than 5 or 20 hosts, respectively. Notably, the transition 'threshold' from one to two MAs (5 hosts) does not coincide with the respective threshold found in the experiment of Section 5.4.1 (see Figure 5.10), which is 13 hosts. This is again due to the time spent on interacting with the managed devices, which is negligible in the latter case. The curves corresponding to SNMP polling latency are also drawn in Figure 7.16, depicting the improved scalability offered by MA-based solutions (SNMP response times for network sizes larger than 10 devices have been extrapolated).

Depending on the managed network size, the optimum number of domains (number of MAs working in parallel) can be determined from the minimum point of the corresponding curves of Figure 7.16b. Being consistent with the measurements presented in Section 5.4.1, the results indicate that assigning very small or large numbers of devices to each MA is unlikely to optimise the overall response time. Thus, for a network of 30 devices, response time is minimised when partitioning the network in 3 domains, i.e. assigning 10 NEs to each MA.

7.6.1.2. SNMP Table polling

One of the main advantages of the SNMP table polling application introduced in Section 7.4 has been claimed to be the improved consistency of the retrieved table values. In this section, we present the results of a simple experiment that proves the validity of that argument. In particular, we compare the time needed to retrieve the contents of several MIB-II tables through:

- remote SNMP interactions, where each table row is retrieved by a single `get-next` request;
- an MA-based approach, where an MA locally interacts with the SNMP agent and obtains table snapshots through successive `get-next` requests.

In that latter approach, we do not measure the MA migration delay but only the time needed to complete the management operation. The performance of the two approaches has been compared for the retrieval of `ifTable` that includes 2 rows and 22 columns (2×22), `ipAddrTable` (2×5), `ipRouteTable` (7×13), `tcpConnTable` (46×5), `udpTable` (18×2).

The results are illustrated in Figure 7.17 and analytically presented in Table 7.6, which also reports the improvement in latency when using the MA-based approach instead of traditional SNMP table polling. As expected, the retrieval latency gain is higher for tables with large number of rows (one `get-next/response` pair of packets is exchanged for every row).

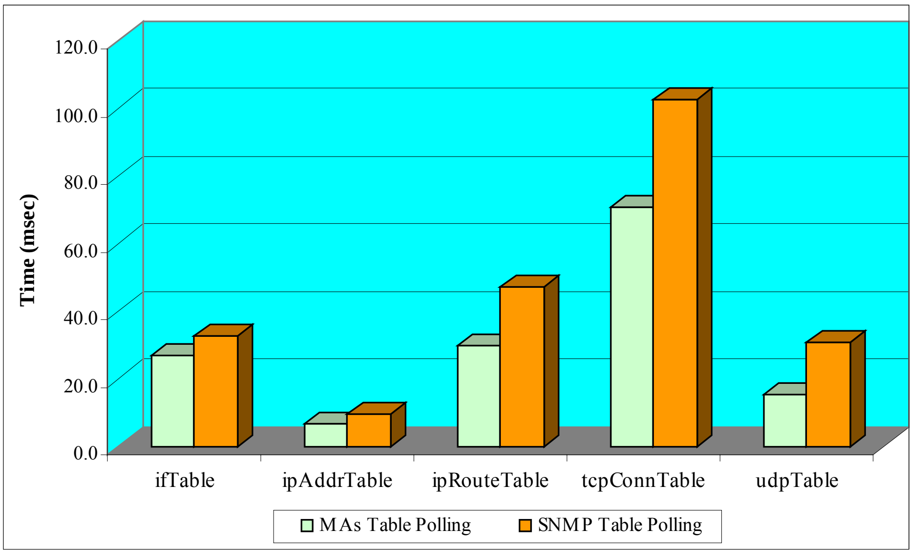


Figure 7.17. SNMP vs. MA-based table polling time measurements

SNMP Table Polling Time (msec)

		ifTable (2×22)		**ipAddrTable** (2×5)		**ipRouteTable** (7×13)		**tcpConnTable** (26×5)		**udpTable** (18×2)	
		Average	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**
Polling method	**MA Table (T_{MA})**	26.9	15.0	6.6	7.1	30.1	16.8	71.0	35.3	15.7	10.1
	SNMP Table (T_S)	37.2	26.0	9.7	11.3	47.2	26.9	102.5	30.6	30.7	38.9
Improvement	**$(T_S - T_{MA}) / T_S$ (%)**	27.8	-	31.6	-	36.3	-	30.7	-	49.1	-

Table 7.6. Latency gain of MA-based against SNMP table polling

The adoption of the MA-based approach certainly does not result in a drastic reduction of the overall latency. However, the time gain is expected to be amplified when: (a) retrieving MIB tables with larger number of rows (e.g. large routing tables), or (b) polling managed devices separated by low-bandwidth links from the manager host. In the latter case, the MA-based approach diminishes the length of the management 'control loop', resulting in low latency per `get-next` request, which is negligible compared to remote network interactions.

Unfortunately, this cannot be pictured in Figure 7.17, where SNMP operations are performed over a high-speed LAN.

7.6.1.3. SNMP Table filtering

The last set of response time experiments aims at evaluating the performance of the table filtering applications described in Section 7.5 and comparing it against SNMP. In particular, two filtering applications have been considered. In the first, a variable number of hosts (one to ten) is polled, retrieving the `ifTable` row corresponding to the network interface that received the maximum number of bytes (max `ifInOctets`). In the second, the contents of polled devices `tcpConnTable` are filtered so as to obtain the table rows for which `tcpConnState` = 'timeWait'.

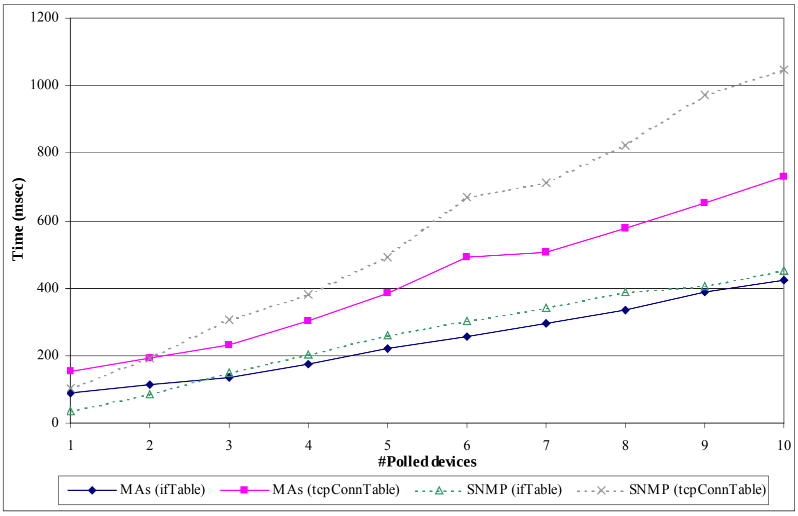


Figure 7.18. Response time performance comparison of SNMP vs. MA-based approach for table filtering applications

With the MA-based approach, a single MA object is launched from the manager platform and sequentially visits every managed device. At each host, the MA obtains a snapshot of the requested MIB table, stores the table rows satisfying the filtering pattern and subsequently migrates to the next host. Our experimental testbed comprises exclusively Windows NT machines, which include only *two* interfaces. In the first application, only one of the two `ifTable` rows is returned to the manager, whilst in the second MAs store an average of 5 out of 28 `tcpConnTable` rows. When applying the SNMP model, table snapshots are first obtained through remote synchronous interactions, with the table contents filtered following the retrieval completion. The results presented in Figure 7.18 and Table 7.7 indicate a slight precedence of the MA-based approach over the SNMP implementation. Regarding the SNMP timings, the processing time needed by the manager application to filter the retrieved tables and obtain the requested high-level information for each of the polled devices has not been measured; the corresponding time measurements are, therefore, lower than the real polling response times.

The employment of GnG polling scheme is expected to further reduce the overall response time of MA-based polling, although relevant experiments have not been conducted. It should be emphasised though, that the primary goal of table filtering applications has been to reduce the network overhead and *not* the response time. Network overhead measurements will be presented in the following section.

SNMP Table Filtering Time (msec)

#Polled Devices	MAs (ifTable)		SNMP (ifTable)		MAs (tcpConnTable)		SNMP (tcpConnTable)	
	Average	SD	Average	SD	Average	SD	Average	SD
1	90.1	18.8	37.2	26	152.9	29.9	102.5	30.6
2	113.0	18.15	83.7	22.1	193.2	23.6	190.8	31.1
3	133.9	13.5	150.2	29.5	230.6	19.9	307.0	18.3
4	175.2	35.1	203.4	30.1	300.9	41.1	380.7	28.6
5	221.7	37.35	260.9	29.5	383.7	34.9	490.9	41.7
6	255.3	51.3	302.1	33.7	490.0	30.6	670.1	55.3
7	294.3	45.15	340.5	40.2	505.1	29	711.1	37.7
8	335.3	27.9	387.2	48.4	575.5	44.2	824.3	42.7
9	389.5	68.25	404.4	39.1	651.3	38.8	970.4	66.3
10	422.9	33.15	451.0	40	728.2	37.4	1045.7	29

Table 7.7. MAs vs. SNMP table filtering times

7.6.2. Network Overhead

The main motivation for developing the introduced applications has been to improve management scalability, mainly in terms of network overhead. In this section, we present the results of a bundle of experiments, which aim at evaluating the applications performance and compare it against that of the centralised model (SNMP). We have examined two separate case studies: In the first, the management of a medium-sized LAN is considered. For simplicity reasons, we assume that all the managed NEs and the manager station are connected in a single network segment. In the second, we consider a more general case, whereby the managed network comprises three separate segments, one of which is connected to the manager station through a slow WAN link. It should be noted that the networking environments corresponding to each of the aforementioned case studies are *not* real. Yet, by measuring the size of the employed MAs code, their initial state size and the state size increment resulting when encapsulating specific data samples, the network overhead incurred when employing each of the introduced applications can be accurately estimated.

7.6.2.1. LAN Environment Case Study

In the first case study, we consider a LAN environment of 50 managed devices. The network overhead imposed by the three proposed applications has been compared against the AdventNet SNMPv1 implementation.

The SNMP `get` request/response message has been measured 90 bytes, on average (at the MAC layer), while every extra value included in the SNMP packet's varbind list represents an additional overhead of 17 bytes, on average. Table 7.8 summarises all the MA attributes needed to evaluate the network overhead of the introduced applications. The values reported in the table have been measured using the *Windump* network analyser [Windump].

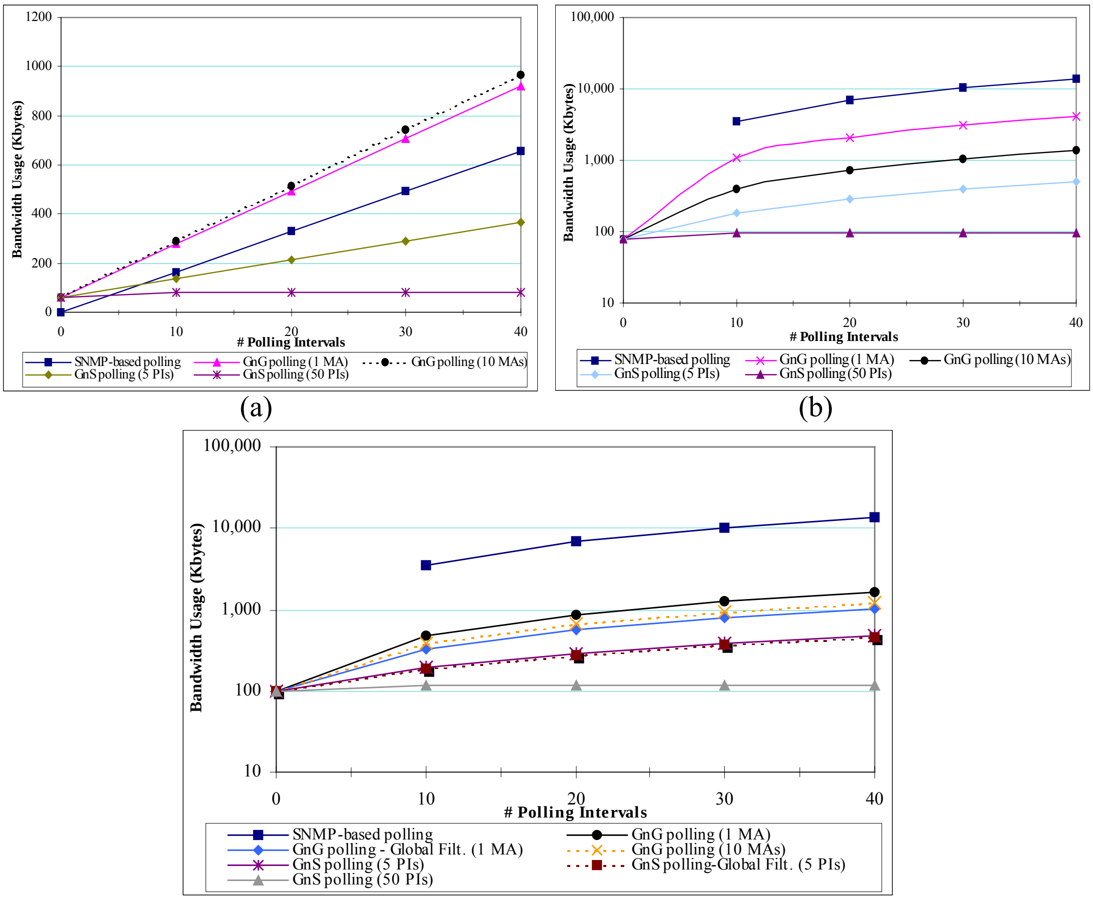
	HF Evaluation	**SNMP table polling**	**SNMP table filtering**
Compressed code size (in Kbytes)	1.25	1.36	1.95
Compressed (initial) state size (in bytes)	381	384	447
State size increment per sample (in bytes)	2 (one extra value)	68 (8×21 extra values)	13 (1×21 extra values)

Table 7.8. Attributes of the MA classes corresponding to the three proposed applications

Figure 7.19a compares the performance of SNMP-based polling against the MA-based approach, when the calculation of the HF appearing in Eqn. (7-3) is considered. Despite the object aggregation, the GnG scheme does *not* outperform centralised polling (the performances of these two approaches are expected to converge for larger number of aggregated values). In addition, the segmentation of the managed network into *ten* domains (employing 10 MAs in GnG polling) does not seriously effect bandwidth consumption, while reducing the overall response time (see Section 7.6.1.1). The situation improves with GnS scheme, which becomes more attractive as the delivery frequency decreases. It is noted that the starting point for GnG/GnS polling is at 62.5Kbytes, representing the overhead incurred for broadcasting the compressed MA code to all NEs (50×1.25Kb).

Figure 7.19b, drawn on logarithmic scale, compares our table polling method against the traditional approach, when considering the retrieval of an interfaces table consisted of *eight* rows (8×21 entries). We assume that each `get-next` request retrieves an entire table row. The MA-based approach clearly surpasses SNMP-based polling due to the lightweight data 'encoding' method employed, data compression, and the fact that the *whole* table is wrapped into the MA's state before delivered to the manager. It is worth noting that the network partitioning into *ten* domains provides better results in this case. This is explained by the rapid growth of the MA's state encountered when a single MA object is responsible for polling all

NEs (the MA state size is increased by 68 bytes each time a NE is visited, as shown in Table 7.8).



(a)

(b)

(c)

Figure 7.19. Bandwidth consumption of SNMP-based polling against the proposed MA-based applications: (a) HF computation, (b) SNMP table polling, and (c) SNMP table filtering

The same experiment is repeated for the table filtering application, with the pre-eminence of the MA-based against the SNMP approach being more distinct (see Figure 7.19c). In this scenario, MAs bring back only the most heavily loaded interface (from each host). It should be emphasised that global filtering improves the framework's performance both in GnG and GnS cases, since it keeps the size of the MAs state constant (in this case, the most heavily loaded *network* interface is returned). The bandwidth usage is reduced even further when single table entries (instead of entire rows) are requested (this is not shown here). The results illustrated in Figure 7.19 are also analytically presented in Table 7.9.

Network Overhead - Health Function Computation (Kbytes)

		SNMP	GnG (1 MA)	GnG (10 MAs)	GnS (5 PIs)	GnS (50 PIs)
#Polling Intervals	0	0	62.5	62.5	62.5	62.5
	10	164.1	277.2	288.7	137.9	81.1
	20	328.1	491.8	514.8	213.3	81.1
	30	492.2	706.5	741	288.7	81.1
	40	656.3	921.1	967.2	364.1	81.1

(a)

Network Overhead - SNMP Table Polling (Kbytes)

		SNMP	GnG (1 MA)	GnG (10 MAs)	GnS (5 PIs)	GnS (50 PIs)
#Polling Intervals	0	0	68.4	68.4	68.4	68.4
	10	3437.5	1068.9	388.6	175.1	87.1
	20	6875	2069.5	708.8	281.8	87.1
	30	10312.5	3070.1	1029	388.6	87.1
	40	13750	4070.7	1349.2	495.3	87.1

(b)

Network Overhead - SNMP Table Filtering (Kbytes)

		SNMP	GnG (1 MA)	GnG - Global Filt. (1 MA)	GnG (10 MAs)	GnS (5 PIs)	GnS Global Filt. (5 PIs)	GnS (50 Pis)
#Polling Intervals	0	0	97.7	97.7	97.7	97.7	97.7	97.7
	10	3437.5	482.1	326.8	378.6	191.3	186.2	119.5
	20	6875	866.6	555.9	659.6	285	274.8	119.5
	30	10312.5	1251.1	785	940.5	378.6	363.4	119.5
	40	13750	1635.6	1014.1	1221.5	472.3	452	119.5

(c)

Table 7.9. Network overhead estimation for SNMP-based polling, against the proposed MA-based applications: (a) HF computation, (b) SNMP table polling, and (c) SNMP table filtering

7.6.2.2. LAN-WAN Environment Case Study

In the second case study, we investigate performance issues in a generic topology scenario, whereby the managed network comprises three separate segments, one of which is connected to the manager site through a low-bandwidth WAN link. Such topology is suitable to evaluate the applicability and the management cost savings achieved when deploying the introduced applications, using as underlying management platform the adaptive hierarchical MA-based framework described in Chapter 6. The test network is shown in Figure 7.20; the network domain margins are depicted by the dotted curved lines, while an indicative label is displayed below the icon of the hosts where Mobile Distributed Managers (MDM) execute.

The quantitative model presented in Section 6.4 has been applied to that test network. Referring to this particular topology, we assign the cost coefficients the following values:

$k_{S_0,S_0} = k_{S_1,S_1} = k_{S_2,S_2} = 1$, $k_{S_0,S_1} = 5$ and $k_{S_0,S_2} = 50$. Coefficient values are chosen so as to reflect the bandwidth availability on the links they correspond to.

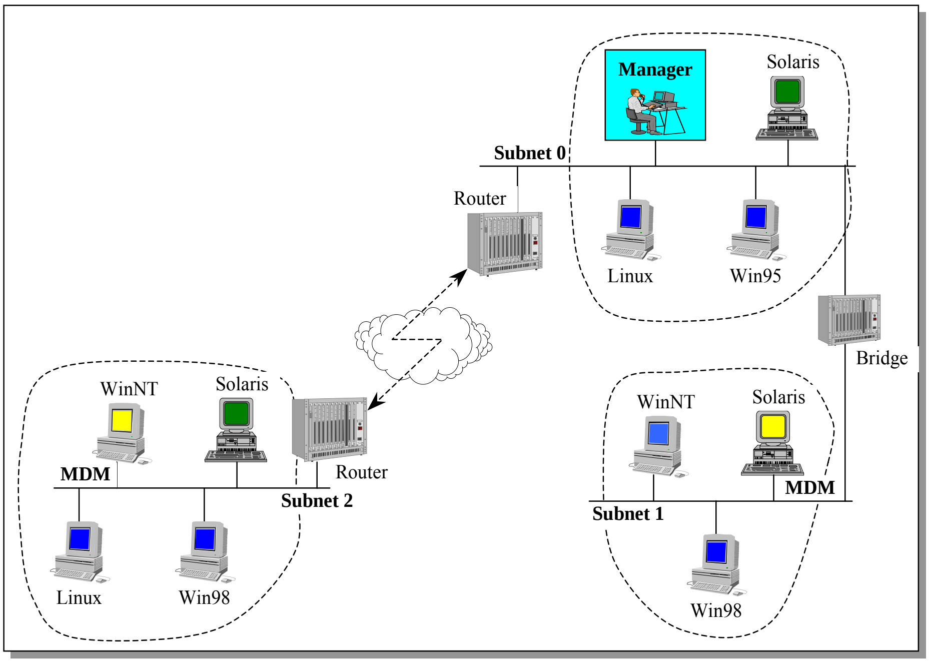


Figure 7.20. LAN-WAN environment test network

The suitability of our hierarchical model has been tested by considering the table filtering application as case study. According to the measurements mentioned in the previous section, the following variables are assigned the values: S_{req} = 90 bytes, ΔS_{req} = 17 bytes, C = 1.95 Kbytes, and ST_0 = 447 bytes. Equations (4-6), (5-2), (5-3), (6-2)-(6-6) are applied to compare the performance of SNMP polling against that of MA-based flat and hierarchical management in terms of the overall cost, as shown in Figure 7.21, drawn on a logarithmic scale. The functions defining the cost of MA-based flat management represent special cases of those developed for hierarchical management.

The same filtering application examined in the preceding section is considered here, i.e. information referring to the most heavily loaded interface of each host is obtained. Using such filtering pattern improves system scalability, due to the low selectivity ratio σ achieved over the acquired data. In particular, the MA state size increases by only 13 bytes for each visited host ($\sigma \times b = 13$ in Eqn (4-7)), as showed on Table 7.8. We also assume that management data are gathered by the MDM and delivered to the manager at regular intervals (in this case, the delivery frequency is F_{del} = 10 PIs). The MDM that runs on Subnet 2 for instance, having to poll 4 devices, it gathers D = 4_devices × 13 bytes/device = 52 bytes every PI and delivers through a Remote Method Invocation (RMI) to the manager a total of $F_{del} \times D = 10 \times 52 = 520$ bytes every 10 PIs.

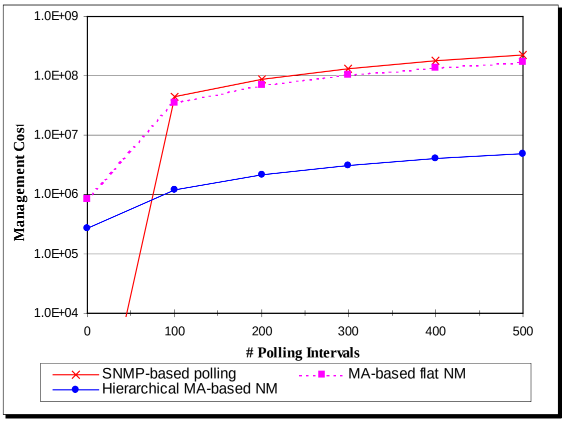


Figure 7.21. Management cost of hierarchical framework against SNMP and flat MA-based polling

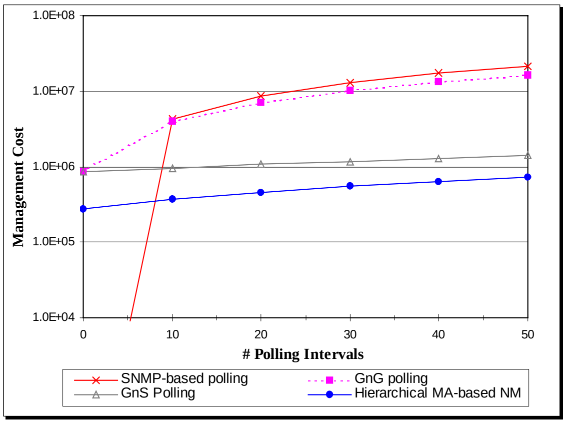


Figure 7.22. Comparison of the management costs for hierarchical, SNMP-based, GnG and GnS polling

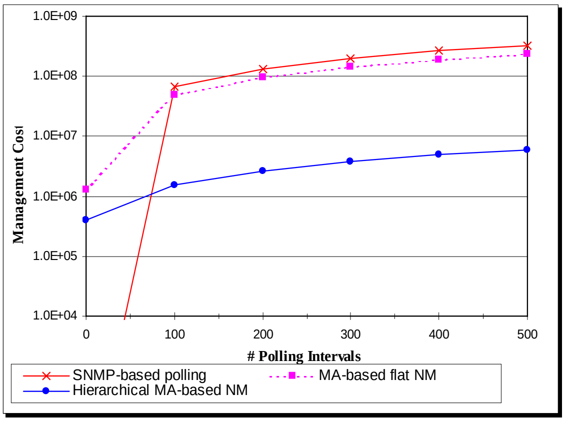


Figure 7.23. Comparison of the management costs when the cost coefficients are proportional to the inverse of link bandwidth

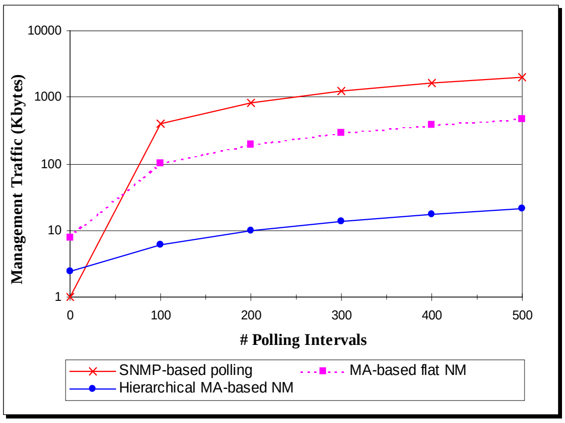


Figure 7.24. Bandwidth usage of the WAN link imposed by hierarchical, SNMP-based and flat MA-based management

Clearly, the introduced hierarchical architecture gives rise to a remarkable reduction of management cost, while the cost of flat management is surpassed by that of centralised polling only after the first *70* PIs. It is also noted that the starting point for the cost induced by the hierarchical infrastructure is much lower than the equivalent of flat management, due to the lightweight "tree multicasting" code distribution scheme, described in Section 6.3.5.

It should be also emphasised that for both SNMP and MA-based flat management, whenever a packet or an MA object is sent through a link, the traffic associated with the actual transfer affects both the network segments attached to that link. For instance, if an MA was to poll all the devices of Subnet 1 and then (on its i^{th} hop) move to Subnet 2 to continue its execution, the migration cost for the transfer between Subnet 1 and Subnet 2 would be: $\left(k_{S_1,S_0} + k_{S_0,S_0} + k_{S_0,S_2}\right) * ST_i$. In contrast, hierarchical MA-based management relaxes the network from this unnecessary burden, as after the MDMs deployment, the management traffic is localised thereby minimising the use of interconnecting links.

As shown in Figure 7.22, the introduction of GnG polling (see Section 5.2.1), where a single MA is assigned to each of the management domains (network segments), slightly improves the performance of flat model in terms of management cost, while it can greatly reduce the overall response time, as evidenced in Section 7.6.1.1. We have also measured the management cost for GnS polling (see Section 5.2.2), where the MAs are assumed to deliver collected data every *10* PIs. Although the cost is reduced compared to GnG scheme, it still does not fall below that of hierarchical management. In addition, GnS polling cannot be utilised in time-critical applications.

Figure 7.23 illustrates the same comparison as Figure 7.21, with the difference that the cost coefficients are now proportional to the inverse of link bandwidths. Assuming that all the subnets are 10 Mbps Ethernet and that the bandwidths of the links connecting Subnet 0 to Subnet 1 and Subnet 2 are 10 Mbps and 64 Kbps respectively, the cost coefficients become: $k_{S_0,S_0} = k_{S_1,S_1} = k_{S_2,S_2} = k_{S_0,S_1} = 1$ and $k_{S_0,S_2} = 156.25$ (when multiplied with a normalising factor $c_{nor} = 10^7$). The increase of the WAN link coefficient value amplifies the separating gap between the cost of hierarchical management and these of SNMP-based and flat management, although this is not 'visible' due to the logarithmic scale.

Figure 7.24 focuses on the traffic generated from each of the examined models on the WAN link connecting Subnet 0 to Subnet *2*. Again, the hierarchical framework outperforms both flat MA-based and SNMP management with sufficient distinct. In particular, following bytecode distribution and MDM deployment, the hierarchical framework uses the WAN link only to deliver the statistics to the manager host, every *10* PIs. In contrast, SNMP heavily utilises the link to broadcast request messages and receive back the associated responses, while in flat management an MA object traverses the link at least twice in every PI, provided that MAs itineraries are optimised so as to poll the remote LAN hosts in sequence. The results illustrated in Figure 7.21-Figure 7.24 are detailed in Table 7.10.

Although our prototype has not yet been tested in a large enterprise network environment, it is expected to scale well, especially when combining our hierarchical framework with GnG/GnS polling schemes, as discussed in Section 6.2. For instance, MDMs could be easily coupled with GnG scheme: after deploying the MDMs to their remote domains (possibly comprising large numbers of hosts), these could launch a sufficient number of MAs per PI in order to reduce the overall response time. Alternatively, coupling the hierarchical model with GnS polling would minimise data transfers within the individual domains.

Management Cost - Fixed Cost Coefficients

#Polling Intervals	SNMP	MA-based flat NM	GnG	GnS	Hierarchical NM
0	10	848640	848640	848640	273467.4
100	44200000	35357340	31978140	1954560	1201967.4
200	88400000	69866040	63107640	3060480	2130467.4
300	132600000	104374740	94237140	4166400	3058967.4
400	176800000	138883440	125366640	5272320	3987467.4
500	221000000	173392140	156496140	6378240	4915967.4

(a)

Management Cost - Cost Coefficients Dependant on Link Bandwidth

#Polling Intervals	SNMP	MA-based flat NM	Hierarchical MA-based NM
0	10	1273958.4	401155.95
100	66352000	49078158.4	1533430.95
200	132704000	96882358.4	2665705.95
300	199056000	144686558.4	3797980.95
400	265408000	192490758.4	4930255.95
500	331760000	240294958.4	6062530.95

(b)

Network Traffic over the WAN link (Kbytes)

#Polling Intervals	SNMP-based polling	MA-based flat NM	Hierarchical MA-based NM
0	1.0	7.8	2.4
100	406.3	100.2	6.2
200	812.5	192.6	10.0
300	1218.8	284.9	13.8
400	1625.0	377.3	17.6
500	2031.3	469.7	21.4

(c)

Table 7.10. (a) Management cost of SNMP against MA-based flat management, GnG, GnS and hierarchical MA-based management for fixed link cost coefficients, (b) Management cost comparison when cost coefficients are equal to the inverse of link bandwidth, (c) Network traffic over the WAN link for SNMP, flat and hierarchical MA-based management

7.7. CONCLUSIONS

We have presented three novel applications of MAs on network monitoring and performance management. The applications, which have been developed on the top of the lightweight MAP described in Chapters 4-6, address the scalability limitations of centralised NSM that become significantly more pronounced when transfers of bulk network monitoring data are considered (see Section 7.2). In particular, MAs have been utilised to: (i) aggregate several MIB values into more meaningful network health indicators (HFs), (ii) acquire SNMP tables snapshots, and (iii) filter SNMP tables contents applying complex filtering expressions.

Delegating HFs to distributed MAs enables direct observation of network behaviours at sufficient precision. By computing health indicators locally, vast amounts of real-time data can be significantly reduced, while HF configurations may be dynamically updated to reflect varying behaviour patterns at different systems or times. Although the idea of using HFs to semantically compress management data is not new (see [GOL93a]), here we exploit the powerful MA paradigm to compute HFs in a more flexible and dynamic manner.

MAs have been also used to efficiently retrieve SNMP tables, where table snapshots are obtained through local interactions and sent to the manager through single transfers. In addition to diminishing the number of remote network interactions, this approach improves management scalability even further through performing more efficient data 'encoding' and data compression, whilst offering a more consistent view of retrieved table contents by decreasing the length of the management 'control loop'. Atomic SNMP table views are particularly useful in a variety of applications aiming at investigating transient problems of short duration or identifying trends on changing networking conditions.

The observation that only a small portion of retrieved SNMP contents is typically useful to management applications, led to the development of a third -and more complex- application. In particular, the ability of MAs to dynamically delegate management intelligence to managed devices has been exploited to perform intelligent filtering of SNMP tables and enable delivery of high-level, pre-processed information to the manager platform. A variety of textual and arithmetic operators are provided to facilitate the exclusion of table values that do not satisfy pre-determined filtering criteria. Decentralised table filtering may lead to significant management cost savings, especially when highly selective filtering patterns are considered, while relaxing the manager platform from processing huge amounts of data.

We have also exploited MAs ability to realise multi-node itineraries to introduce the concept of domain/global filtering, where MAs use the knowledge/information already collected to perform a superjacent level of data filtering. Global filtering approach defends the use of the MAs' mobility feature in NSM applications, especially in cases where a 'global view' of the managed network is required, i.e. when data obtained from different sources need to be correlated. Although it cannot be considered as 'killer application', it certainly represents a case where the use of multi-hop agents is the most appropriate delegation technique, if not always, at least in many scenarios. That fact has also been acknowledged in [MAR01], which presents a critic perspective on the current status of MA technology with special focus to NSM and cites our application as one of the few where the use of agent mobility is justified.

The experimental results presented in Section 7.6 have demonstrated that the introduced applications are competent in terms of response time, compared to a Java-based SNMP stack implementation (in fact, a slight performance precedence has been indicated). In addition, they confirmed a significant improvement on traffic overhead when testing the proposed applications in realistic management scenarios and comparing them against traditional centralised polling.

As a final point, it should be stressed that MA-based approaches to distributed management applications are neither a panacea nor the only possible solution to distribution problems. Many of the ideas proposed in this chapter could be realised using the concepts of OSI-SM management functions or, alternatively, by modifying the SNMP framework (e.g. by incorporating dynamically configured data filtering capabilities within SNMP agents).

CHAPTER 8

SUMMARY, CONCLUSIONS & DIRECTIONS FOR FUTURE WORK

In this final chapter we bring together the work described in the previous chapters of this thesis. Section 8.1 explains the significance of the thesis in terms of its main contributions and summarises the main findings. Section 8.2 presents the conclusions drawn through the research experience gained, regarding network monitoring applications in which agent mobility can be effectively used. Section 8.3 identifies areas in which this work could be developed further.

8.1. SUMMARY OF MAIN CONTRIBUTIONS

The main objective of this thesis has been to assess the values and weaknesses of Mobile Agent (MA) technology form the management viewpoint and identify ways for the effective use of this technology in distributed management applications. That involved the design and implementation of MA platform (MAP) expressly oriented to Network & Systems Management (NSM), and the development of three network monitoring and performance management applications implemented on the top of our framework. The research contributions of this thesis have been outlined at the end of Chapters 4-7. We re-iterate through the main findings below.

The contributions and findings related to the design and implementation of our MA-based management framework are summarised in the following:

- The MA framework introduced in Chapter 4, is characterised by a *lightweight* design that makes it particularly suitable for developing MA-based applications with minimal impact on network and system resources. That has mainly been achieved through designing a code distribution scheme that involves the transfer of MA bytecode to managed devices at the MAs' creation time, with only MAs state being transferred following that. This is in contrast with the majority of MAPs, which enforce the transfer of both the MA code and state in every agent migration. This code distribution model has been refined in Chapter 6, by taking advantage of remotely located Mobile Distributed Managers (MDM) to implement a *tree multicasting* code distribution scheme. Mobile Agent Servers (MAS) have also been designed so as to have a lightweight footprint on hosting devices.

- Being implemented in Java, the framework is *portable* across all platforms supporting the Java Virtual Machine (JVM), addressing the requirement for integrated management.
- *Integration with legacy systems* is achieved through an MA-to-SNMP gateway integrated within the MAS, which allows incoming MA objects to locally interact with SNMP agents.
- A set of *security features* has been incorporated, shielding network devices against malicious MAs attacks and protecting management data from eavesdropping. In particular the Security Component (SC), an integral component of the MAS, provides authentication and access control (authorisation) services, whilst the data carried by MAs can optionally be encrypted.
- *Fault-tolerance* issues have also been addressed, covering scenarios where a host included in the MAs' itinerary (or the manager) fails or an interconnecting link breaks.
- The MA class versioning problem has been addressed through a *customised MA ClassLoader* (MACL), which allows distinguishing between different versions of the same MA class, thereby enabling modifications of MA-based NSM tasks at runtime.
- A Mobile Agent Generator (MAG) tool has been implemented to *automate the introduction/customisation of service-oriented MAs* through a user-friendly Graphical User Interface (GUI).
- Additional features include: (a) design of GUIs offering Management Information Base (MIB) browsing, data visualisation, management tasks customisation, etc.; (b) support for Remote Method Invocation (RMI) interactions between the manager and the managed devices; (c) visual profiling and active control over remotely executing MA threads; (d) support for MA transfers over TCP or UDP protocols, with the choice made by the administrator depending on the application; (e) management data compression.
- Experimental results revealed that our framework marginally outperforms Java RMI both in terms of network overhead and response time when considering simple manager-managed systems interactions. However, a detailed investigation of the factors affecting MA-based NSM scalability indicated an *exponential growth of management cost with the managed network size when using a single multi-hop MA* to collect data from a number of managed devices ('flat' MA-based management); that represents a non-scalable solution, especially when MAs are not highly selective. This conclusion dictates that *MA itineraries length should be limited* and led to the optimisations described in Chapter 5.

- In particular, Chapter 5 introduced *two complementary polling schemes* that aim at answering the scalability problems of flat MA-based management. In particular, *Get 'n' Go* (GnG) polling is suitable for collecting real-time data and performing short-term control and configuration tasks on multiple network elements (NE). Response time is minimised by relying on a number of MAs to carry out distributed management tasks in a parallel fashion. In addition, when considering networks with large numbers of managed hosts, the network overhead of flat management is also reduced due to the limited number of hops corresponding to each MA (the overall amount of data accumulated within MAs state is reduced). Regarding the off-line analysis of management data, we proposed the *Go 'n' Stay* (GnS) polling scheme. In this approach, single-hop agents are dispatched to managed devices and collect a large number of data samples before delivering performance reports to the manager, thereby leading to drastic reduction of MA transfers. Collected data can be delivered either by the originally deployed MAs, clones of these MAs or RMI invocations; the second approach has been found to be the most cost-effective solution. In the GnS approach, MAs-manager communication can also be event-driven, with the manager notified only when specific performance thresholds are crossed. Concluding, the selection of the appropriate polling scheme is application-dependent, with the management task execution period and the type of management data to be collected being the main factors. Experimental analysis of GnG scheme has shown that there is an optimal number of MAs that minimises the overall latency and network overhead, depending on the managed network size; GnS polling overhead is reduced for lower data delivery frequencies.
- In Chapter 6, we introduced further extensions, devising a *flexible and adaptive hierarchical MA-based management model*, which meets the requirements for the management of dynamic, large-scale enterprise networks. That model incorporates mobile mid-level manager entities (MDMs), transparently deployed to remote network domains to take over their management responsibility and localise the associated traffic. Our infrastructure offers adaptability to changing networking environments and defines concrete policies regarding network segmentation in management domains and MDMs deployment, explicit determination of domain boundaries, etc.
- To minimise network overhead on remote domains, *MDMs rely on MAs for the data collection process*. Depending on the application, these can either be single-hop or multi-hop agents. MDMs may also move within their managed domain to ensure *balanced distribution of processing and memory load* over managed systems. In particular, MDMs periodically inspect the resources availability on managed nodes and choose to move and resume execution to the least loaded host. Finally, *fault tolerance* issues are also addressed,

securing that distributed MDM objects continue to perform their decentralised tasks even when the communication with the manager platform is not feasible. When considering the management of remote domains, empirical results demonstrated that the proposed architecture outperforms alternative solutions, both in terms of the overall management cost and the bandwidth usage of low-bandwidth WAN links.

- It is emphasised that the management models proposed throughout Chapters 4-6, are supplemented by analytical *quantitative evaluations* and *experimental results*, which quantify their performance in terms of the latency and network overhead incurred when used for realistic management operations.

The contributions and findings related to the network monitoring and performance management applications described in Chapter 7, are listed below:

- The first application involved the *computation of health functions* (HF) by MAs, enabling the semantic compression of several MIB variables into a single system indicator. That allows the direct observation of managed systems health and obviates the need for transferring vast amounts of data over the network. HF configurations may be dynamically updated to reflect management needs at different systems or times, while HF values can optionally be delivered to the manager only when pre-determined thresholds are crossed. Either single-hop or multi-hop agents can be used to compute HFs. In LAN environments, the latter perform worse than SNMP, while single-hop agents (GnS polling) offer the most scalable solution. In LAN-WAN management scenarios, multi-hop MAs outperform SNMP as they reduce the usage of expensive WAN links. However, in that scenario, the MA-based hierarchical model improves management scalability even further.

- The second application involved *efficient SNMP table retrievals*. MAs have been used to obtain table snapshots through local interactions with SNMP agents; table snapshots are then delivered to the manager through single transfers. That scheme improves management scalability as it reduces the number of network interactions and applies data compression, whilst ensuring improved consistency of retrieved table contents. A variety of applications can benefit from table snapshots to investigate transient problems of short duration or identify trends on changing networking conditions. In cases that views of SNMP tables including large amounts of data are requested, single multi-hop agents represent an non-scalable approach as the state of these MAs would rapidly grow; in that case, using multiple MAs (GnG polling) can offer performance benefits, yet, GnS polling provides the most scalable solution.

- The third, and most complex, application exploits the ability of MAs to perform intelligent *filtering of SNMP tables* and enable delivery of high-level, pre-processed information to the manager platform. Filtering patterns can be defined/reconfigured at runtime, using a variety of textual and arithmetic operators, ensuring the transfer of only those table values that satisfy specified filtering criteria. Table filtering operations may lead to significant cost savings, especially when highly selective filtering patterns are involved. Furthermore, multi-hop agents have been used to *correlate* the data collected during their itinerary and offer domain/global views of managed devices. Empirical results have shown that MA-based approaches outperform SNMP with sufficient distinct, especially when using our flexible hierarchical model for table filtering applications in a LAN-WAN environment.

- An important contribution of the introduced applications relates to a factor that currently prevents the acceptance of MAs, whose nature is less technological. MAs will be adopted only when a sufficient body of literature will provide incontrovertible evidence about *when*, *where* and *how* they are useful. The proposed applications serve as case studies in which the implications of using MAs in a real application domain can be *thoroughly analysed on a quantitative and experimental basis*; such case studies are still very rare. The MA research community must put more effort into validating its own outcomes, in order to gain credibility outside.

8.2. APPLICABILITY OF AGENT MOBILITY IN MONITORING APPLICATIONS

One of the main objectives of the research work described in this thesis has been to identify efficient mobility patterns for management applications. That involved the investigation of single-hop and multi-hop mobility, with the latter representing the most common use of MAs [KU97, SAH98, CHI99, ZAP99, ELD99, PUL00b]. Generally, multi-hop agents can offer flexibility and performance benefits over single-hop agents in the scenarios described in Section 3.4. In the management context and particularly on network monitoring and performance management, we have identified the following areas in which multi-hop agents can enhance flexibility and scalability:

- *Collection of real-time data*: Multi-hop agents represent an efficient approach for real-time monitoring operations, i.e. when response time restrictions apply. Specifically, when the latency involved in MA migrations is relatively small (i.e. lightweight, efficient agent transfer protocols are used), multi-hop mobility may significantly improve the efficiency of management data collection operations, compared to the single-hop mobility approach, in which an MA object is deployed to every device. However, this performance gain is conditional to the following: (a) short-term monitoring tasks performed over large sets of hosts should be involved, especially when these hosts reside in remote subnets (in such case, each MA would typically execute for a time negligible with respect to its deployment time); (b) MAs should be able to perform semantic compression of collected data (otherwise their state size will rapidly grow, affecting both the network overhead and response time of the monitoring task). In addition, the latency of time-sensitive management operations can be further reduced using the GnG polling scheme.

- *Data correlation*: The ability of MAs to realise multi-hop itineraries can be exploited to perform data correlation, bringing forth the concept of global filtering. In particular, MAs deliver to the manager a domain/global-level view of managed devices and obviate the need for further processing/correlation of data either by the manager or a mid-level entity. That saves the manager from considerable processing burden and provides a more even distribution of computational load. To the best of our knowledge, this is the only scenario in which MAs employed in monitoring applications use the information already collected to perform a superjacent level of data filtering. Hence, the ability to perform data correlation can be used as a solid argument in favour of multi-hop mobility in the management domain.
- *Flexible and adaptive hierarchical management*: MAs have been shown particularly suitable to implement flexible mid-level managers (MDMs), able to perform decentralised management tasks. In this context, agent mobility can be effectively exploited to localise management traffic, reduce the dependency on inter-connecting links and perform distributed management tasks without the manager's intervention. Although, in theory, this can be achieved through simply uploading the mid-level manager code to a remote host (i.e. using single-hop agents), multi-hop agents add a new dimension in management flexibility and autonomy. For instance, they can dynamically migrate to another host to optimise the average resource usage on managed systems. Furthermore, they can directly observe changes in topology or traffic characteristics, re-define the boundaries of their assigned domain and create clones to share its management responsibility. Remote cloning offers the advantage of reducing the latency and overhead associated with the deployment of a new mid-level manager entity from a central location [LIO99].

It should be emphasised though that *multi-hop agents do not represent a one-way approach* on MA-based management applications. As explained in Section 3.2.3., multi-hop MAs can be regarded as a 'superset' of single-hop mobility, as they can offer all the functionality provided by the latter, with the additional ability of autonomous migration. As a result, single-hop agents can complement their multi-hop counterparts and provide an effective management delegation mechanism in management scenarios where the latter fail to offer performance gain. In particular, single-hop agents are suitable for dynamically augmenting the management capabilities of NEs, monitoring the performance and health of managed systems over long periods and minimising data transfers by returning QoS alarms and periodic summarisation reports after processing raw management information.

8.3. DIRECTIONS FOR FUTURE WORK

The current prototype of our core MA framework presented in Chapter 4 can serve for structuring scalable, flexible and dynamically customised distributed management operations. The following optimisations could be considered in future extensions:

- The framework can be extended to enable web-based management operations. To achieve that, the front-end of the framework should be designed as an *applet* rather than an *application*, allowing the administrator to remotely control its managed network through loading the management applet on standard web browsers.
- MA transfers can be implemented using Java RMI as ‘transport protocol’ and compared, in terms of latency and network overhead, against MA transfers over TCP and UDP.
- To provide a more complete performance evaluation, the performance of our framework should be compared against that of the Common Object Request Broker Architecture (CORBA). Although in theory CORBA is more heavyweight and slow than Java RMI, the choice of an appropriate implementation may lead to marginal performance gain compared to RMI [BOH00a].
- The integration of a resource accounting tool, such as *Jres* [CZA98] can be considered as a means of controlling the occupation of system resources (CPU, memory, etc) within the Java runtime system and potentially killing mis-behaving MA processes.
- The integration of the SC with the Java Cryptography Architecture (JCA) can also be considered to allow compliance and interoperability with other cryptography implementations. Such integration will incorporate the design of a *provider* [Providers], encapsulating the implementations of the SC component’s authentication and encryption algorithms.
- The framework’s fault tolerance could be improved so as to cover a broader range of fault scenarios in addition to the ones mentioned in Section 4.4.2. For instance, the case where the hosting device of an MA fails while the MA is still executing its task should also be taken into account. Therefore, MAs should transparently resume computations that are affected by system or network interruptions or failures. That can be achieved by periodically saving the MA’s persistent state information, as proposed in [BRU01]. This feature should be optionally used depending on hosting devices storage capacity.

- The functionality of the MACL (described in Section 4.4.3.) can be extended so as to maintain information about the last time each of the existing MA classes has been loaded. For those classes not being used for time exceeding a certain limit, their bytecode will be removed from the MACL's hashtable and their associated class files deleted. That way, the use of local memory and disk space would be minimised. In the case that one of the MAs whose bytecode has been removed visits the same device, MACL would download its code from the manager's Mobile Code Repository (MCR) component. An alternative way for optimising the usage of local memory and disk resources would be to enforce the MAs themselves to inform the managed devices whether they execute a long or short-term management task. In the latter case, the Migration Facility Component (MFC) would request MACL to remove the MA code from the hashtable, as soon as the MA migrates to another host.

- Additional code distribution models need to be investigated and compared against the solution adopted in our prototype. For example, the *code pre-fetching* proposed in [SOA99], whereby the bytecode of an MA object is transferred to the hosts included into the MA's itinerary right before the MA's travel begins.

- Finally, several optimisations could also be performed to maximise the framework's performance with respect to the network overhead and response time associated with MA transfers. In particular, the optimisations suggested in Appendix B considerably reduce the MAs state size and therefore moderate the impact of MA migrations upon network resources. In addition, response time could be further reduced by saving the time needed to create MA objects at the beginning of each Polling Interval (PI) and also the time needed to translate host names to their respective IP addresses. This issue is also discussed in Appendix B.

Regarding the two complementary polling schemes described in Chapter 5, the following optimisations could be considered in future extensions, to further improve their efficiency:

- GnG scheme performance could possibly benefit by applying a variation of the technique described in [BAR99]. Namely, enforce individual MAs to download their state (i.e. their collected data) to the manager station after visiting a fixed number of NEs. That would represent significant gain in network overhead, especially when large amounts of data are collected from each host, as it would prevent MAs state from growing over a certain limit; yet, response time penalties associated with the employment of such scheme should be investigated in detail.

- Optimal itineraries can be designed by implementing the Optimal Itinerary Planning (OIP) algorithm described in Section 5.2.1.2.

- In GnG polling, network segmentation could be automated and dynamically adapted to the managed network, aiming at minimising the response time and/or network overhead depending on the number and physical distribution of NEs.
- The performance of GnS polling can be optimised by eliminating information carried within the MAs' state that is only useful for multi-hop agents, e.g. itinerary.
- Additional experiments should be conducted in large-scale networking environments comprising several remote management domains. That would allow to evaluate the effect of network partitioning in reducing the overall response time over flat MA-based model. Should experiments on such environments are not feasible, 'virtual' remote segments could be created, introducing some fixed time penalty to MA transfers whenever an agent traverses a simulated 'WAN link'.

The functionality of the hierarchical model proposed in Chapter 6 can be further improved through incorporating the following optimisations:

- The topology map GUI can be designed as an expandable graphical component: The map will initially display a representation of the managed network where individual subnets will be illustrated as icons that will expand to provide a detailed view of their included managed devices.
- MDMs deployment Policy 2, described in Section 6.3.3., can be implemented to enable the dynamic placement of MDMs depending on the management cost associated with the management of a remote domain rather than the number of devices included therein.
- The Resources Monitoring Tool (RMT) tool (see Section 6.3.7) can be extended so as to support Windows 95/98 and 2000 operating systems.
- The configuration of the hierarchical model can be saved through serialising the `HierarchicalSettings` object (see Section 6.3.3.) whenever the settings are modified. The serialised information would be stored in a file, used to restore the settings at manager startup. That would obviate the need for the administrator to re-define the hierarchical system's configuration.
- MDMs autonomous decision making can be further enriched: in the current version of our prototype, the decision regarding the segmentation of a management domain (already assigned to an MDM) in two parts, is made by the manager. For instance, should a new device is connected to a segment within the boundaries of the examined domain, its local MAS server will publicise its initialisation through communicating directly with the manager. It is then left to the manager to decide whether the remote MDM can cope with the increased number of managed devices or its domain should be divided in two. As a

future extension, we intend to enforce the MDMs to make such decisions without requiring any communication with the manager. In particular, at the time that an MAS server starts, it will first get (through a network 'map') a list of the hosts residing on the same subnet and then broadcast an "I'm-alive" message to them. Should an MDM executes on one of these devices, it will instantly acknowledge the receipt of the message and append the 'discovered' host to its list of managed devices, otherwise, after a pre-determined time interval elapses, the MAS will send the message to the manager platform. In the former case, if the number of NEs managed by the MDM exceeds the specified limit, the MDM will divide its domain in two parts, create a clone of itself and send its clone to take over the management of the second domain. That approach will increase the ability of the MDMs to make autonomous decisions, while minimising the dependency on links connecting remote subnets to the manager site and also the overhead of MDM deployment.

- The current three-level hierarchical model can be extended to a multi-level hierarchy, whereby MDMs could be supervised by other MDMs residing on higher levels rather than directly by the manager platform.
- The existing prototype allows the manager to download on-the-fly new monitoring task definitions to distributed MDMs. However, the functionality of the MDMs themselves cannot be modified at runtime. Therefore, rather than deploying MDMs with fixed capabilities able to perform only dynamically customisable monitoring tasks, the MDM should also provide an interface allowing the manager to plug-in software components implementing new functionality. That interface could be implemented through the `MdmRmiServer` component thereby enabling the design of MDMs as extensible engines.

The functionality of the monitoring applications described in Chapter 7 can be further enriched through applying the following extensions:

- Management applications often need to retrieve information scattered among several MIB tables, e.g., routing information specific to certain type of interfaces and their current utilisation. For instance, the MIB-II `ipRouteTable` [McC91] keeps track of IP routes, with interfaces information found in `ifTable`. An application may need to correlate routes with interface utilisation for capacity planning purposes. That correlation could be achieved by using the `ipRouteIfIndex` column of `ipRouteTable` as index for the corresponding retrievals from the `ifTable`. Such operations imply *joining* two MIB tables, in a fashion similar to database table joins. Currently the table filtering operations described in Section 7.5 do not support table joins, which could be considered in future extensions.

- The applications described in this chapter considered exclusively filtering of MIB-II objects/tables. Hence, additional applications and filtering patterns should be envisaged and applied to other MIBs. For instance, MAs may work in conjunction with Remote Monitoring (RMON) probes to provide, on demand basis, high-level network-oriented statistics, e.g. an MA could visit all the devices with installed RMON probes and return to the manager information about the least loaded Ethernet segment. This information, retrieved from the RMON MIB [WAD95], could be used for network planning, for example to help the administrator decide where to connect a new managed device so as to achieve even distribution of traffic.

  Another scenario could be to return a number of *N* host pairs that communicate in more frequent basis and therefore generate a substantial amount of traffic (this information can be extracted from the RMON *matrix* group). The administrator might then consider connecting these hosts in the same segment so that the generated traffic will only affect that segment. In the case that no RMON probes are installed, MAs could extract similar information from packet sniffers, like the C-based sniffer tool used in [FER01].

- The latency associated with table retrievals can be reduced by implementing the ‘get-table’ operation through successive `get-bulk` (instead of `get-next`) requests; that would also improve table values consistency. Since the get-bulk operation is only available in SNMP v2c and v3 frameworks, MAs could be instructed to first check the SNMP version of the local agent and then accordingly invoke an appropriate ‘get-table’ method implementation, issuing either `get-next` or `get-bulk` requests.

- Current filtering patterns allow Table Filterer (TF) agents to either obtain *single* column values or the whole table *row*. These patterns should be extended so as to allow encapsulation of an *arbitrary* number of table column values.

- The MAG tool’s functionality can be extended so as to provide the user a ‘library’ of actions to be triggered at the event of specific threshold crossings. Such an extension would enable MAs to perform simple configuration or fault management tasks.

BIBLIOGRAPHY

[ACH97] Acharya A., Ragnganathan M., Saltz J., "Sumatra: A Language for Resource-Aware Mobile Programs", Proc. of the 2nd Int. Workshop on Mobile Object Systems (MOS'97), LNCS vol. 1222, Springer-Verlag, July 1997.

[ACL] FIPA, FIPA ACL Message Structure Specification. Revision XC00061D, August 2000.

[ADA00] Adachi A., De Mondonca M.F., De Souza J.N., "Push Model in Web-Based Network Management", Proc. of the Int. Conf. on Telecommunications (ICT'2000), May 2000.

[AdventNet] AdventNet, http://www.adventnet.com/.

[Aglets] IBM, Aglets, http://www.trl.ibm.co.jp/aglets/.

[AMETAS] AMETAS (Asynchronous MEssage Transfer Agent System), Distributed Systems and Operating Systems research group, Johann Wolfgang Goethe-University, Frankfurt, Germany, http://www.vsb.cs.uni-frankfurt.de/ametas/.

[ANE98] Anerousis N., "An Information Model for Generating Computed Views of Management Information", Proc. of the 9th IFIP/IEEE Int. Workshop on Distributed Systems: Operations & Management (DSOM'98), pp. 169–180, Oct. 1998.

[ANE99] Anerousis N., "An Architecture for Building Scalable, Web-based Management Services", Journal of Networks and Systems Management, 7(1), March 1999.

[APP94] Appleby S., Steward S., "Software Agents for Control", in Modelling Future Telecommunication Systems, Cochrane P. and Mealthey P. Eds., Chapman & Hall, 1994.

[ARN96] Arnold K., Gosling J., "The Java Programming Language", Addison-Wesley, 1996.

[AUS00] Austin C., Pawlan M., "Writing Advanced Applications for the Java Platform", March 2000, http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/index.html.

[BAL97] Baldi M., Gai S., Picco G.P., “Exploiting Code Mobility in Decentralised and Flexible Network Management”, Proc. of the 1st Int. Workshop on Mobile Agents (MA’97), LNCS vol. 1219, pp. 13-26, April 1997.

[BAL98] Baldi M., Picco G.P., “Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications”, Proc. of the 20th Int. Conf. on Software Engineering (ICSE’98), pp. 146-155, April 1998.

[BAR99] Barbeau M., “Modeling and Comparison of Bandwidth Usage of Three Migration Strategies of Mobile Agents”, Proc. of the 1st Int. Workshop on Mobile Agents for Telecommunication Applications (MATA’99), pp. 197-210, Oct. 1999.

[BAU99] Baumer G., Breugst M., Choy S., Magendaz T., “A Universal Agent Platform Based on OMG MASIF and FIPA Standards”, Proc. of the 1st Int. Workshop on Mobile Agents for Telecommunication Applications (MATA’99), pp. 1-18, Oct. 1999.

[BEL99] Bellavista P., Corradi A., Stefanelli C., “An Open Secure Mobile Agent Framework for Systems Management”, Journal of Network and Systems Management (JNSM), 7(3), Sep. 1999.

[BIE97] Bieszczad A., Pagurek B., “Towards Plug-and-Play Networks with Mobile Code”, Proc. of the Int. Conf. on Computer Communications (ICCC’97), Nov. 1997.

[BIE98a] Bieszczad A., Pagurek B. “Network Management Application-Oriented Taxonomy of Mobile Code”, Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS’98), Feb. 1998.

[BIE98b] Bieszczad, A., White, T., Pagurek, B., “Mobile Agents for Network Management”, IEEE Communications Surveys, 1(1), Sep. 1998.

[BOG73] Boggs, J., “IBM Remote Job Entry Facility: Generalize Subsystem Remote Job Entry Facility”, IBM Technical Disclosure Bulletin 752, IBM, Aug. 1973.

[BOH00a] Bohoris C., Pavlou G., Cruickshank H., “Using Mobile Agents for Network Performance Management”, Proc. of the 2000 IEEE/IFIP Network Operations and Management Symposium (NOMS’2000), pp. 637-652, April 2000.

[BOH00b] Bohoris C., Liotta A., Pavlou G., "Software Agent Constrained Mobility for Network Performance Monitoring", Proc. of the 6th IFIP Conf. on Intelligence in Networks (SmartNet'2000), pp. 367-387, Sep. 2000.

[BOH00c] Bohoris C., Liotta A., Pavlou G., "Evaluation of Constrained Mobility for Programmability in Network Management", Proc. of the 11th IFIP/IEEE Int. Workshop on Distributed Systems: Operations & Management (DSOM'2000), pp. 243-257, Dec. 2000.

[BON98] Bonabeau E., Henaux F., Guerin S., Snyers D., Kuntz P., Theraulaz G., "Routing in Telecommunications Networks with Ant-Like Agents", Proc. of the 2nd Int. Workshop on Intelligent Agents for Telecommunication Applications (IATA'98), LNCS vol. 1437, pp. 60-71, July 1998.

[BRU01] Bruce S., Linton D.G., "Evaluating Recovery Protocols for Mobile Agents in Network Management Applications", Proc. of the 7th IFIP/IEEE Int. Symposium on Integrated Network Management (IM'2001), in press, May 2001.

[BUM00] Bumpus W., Sweitzer J.W., Thompson P., Westerinen A.R., Williams R.C., "Common Information Model: Implementing the Object Model for Enterprise Management", Wiley, 2000.

[CAB00] Cabri G., Leonardi Z., Zambonelli F., "Weak and Strong Mobility in Mobile Agent Applications", Proc. of the 2nd Int. Conf. and Exhibition on the Practical Applications of Java (PA-Java'2000), April 2000.

[CAR97] Carzaniga A., Picco G.P., Vigna G., "Designing Distributed Applications with Mobile Code Paradigm", Proc. of the 19th Int. Conf. on Software Engineering (ICSE'97), pp. 22-32, May 1997.

[CAS90] Case J., Fedor M., Schoffstall M., Davin J., "A Simple Network Management Protocol (SNMP)", RFC 1157, May 1990.

[CAS93] Case J., McCloghrie K., Rose M., Waldbusser S., "Manager-to-Manager Management Information Base", RFC 1451, April 1993.

[CAS96] Case J., McCloghrie K., Rose M., Waldbusser S., "Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2) ", RFC 1906, Jan. 1996.

[CCITT92] CCITT, "Maintenance: Telecommunications Management Network - TMN Management Functions", Recommendation M.3400, Oct. 1992.

[CHA97] Chadha R., Wuu S., “Incorporating Manageability into Distributed Software”, Proc. of the 5th Int. Symposium on Integrated Network Management (IM’97), pp. 489-502, May 1997.

[CHE95] Chess D., Grosof B., Harrison C., Levine D., Parris C., Tsudik G., “Itinerant Agents for Mobile Computing”, IEEE Personal Communications, 2(5), pp. 34-49, Oct. 1995.

[CHE98] Cheikhrouhou M., Conti P., Labetoulle J., “Intelligent Agents in Network Management, a state of the art”, Networking and Information Systems, 1(1), pp. 9-38, 1998.

[CHE00a] Cheikhrouhou M., Conti P., Labetoulle J., “Automatic Configuration of PVCs in ATM Networks”, Proc. of the 2000 IFIF/IEEE Network Operations and Management Symposium (NOMS’2000), pp. 535-548, April 2000.

[CHE00b] Cheikhrouhou M., “Software Agents in Network Management”, PhD thesis, Insitut Eurocom/Ecole Polytechnique Federale, Lausanne, Sep. 2000.

[CHI97] Chia T.H., Kannapan S., “Strategically Mobile Agents”, Proc. of the 1st Int. Workshop on Mobile Agents (MA’97), LNCS vol. 1219, pp. 13-26, April 1997.

[CHI99] Chirico M., De Milato F., Scapolla A.M., “Mobile Agents for Secure Management of Networks”, HP Openview University Association (HP-OVUA’99), June 1999.

[Cisco00] Cisco Systems, Release Notes for Cisco IOS Release 12.1(5)DA for Cisco DSLAMs with NI-2, Dec. 2000, http://www.cisco.com/univercd/cc/td/doc/product/dsl_prod/ios_dsl/rel121/relnotes/reln1215.htm.

[Concordia] Mitsubishi Electric, Concordia, http://www.meitca.com/HSL/Projects/Concordia/.

[COR98a] Corradi A., Stefanelli C., “How to Employ Agents in Systems Management”, Proc. of the Third Int. Conf. and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents (PAAM’98), pp. 17-26, March 1998.

[COR98b] Corkery M., “A Review of State of the Art in Mobile Agent Systems”, Technical Report, Dep. Computer Science, National University of Ireland, Maynooth, Ireland, 1998.

[CORBA] CORBA/IIOP 2.4 Specification, Nov. 2000, http://www.omg.org/technology/documents/formal/corbaiiop.htm.

[CZA98] Czajkowski G., Von Eicken T., "Jres: a Resource Accounting Interface for Java", Proc. of the Int. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98), Nov. 1998.

[DAS99] Dasgupta P., Narasimhan N., Moser L.E., Melliar-Smith P.M., "MAgNET: Mobile Agents for Networked Electronic Trading", IEEE Transactions on Knowledge and Data Engineering, 11(4), pp. 509-525, July-Aug. 1999.

[DCOM] Microsoft, "Distributed Component Object Model Protocol -- DCOM/1.0", Nov. 1996, http://msdn.microsoft.com/library/specs/distributedcomponentobjectmodelprotocoldcom10.htm.

[Disman] Bierman A., Greene M., Stewart B., Waldbusser S., "Distributed Management Framework", August 1998, http://www.alternic.org/drafts/drafts-i-j/draft-ietf-disman-framework-02.txt.

[ELD99] El-Darieby, Bieszczad A., "Intelligent Mobile Agents: Towards Network Fault Management Automation", Proc. of the 6th IFIP/IEEE Int. Symposium on Integrated Network Management (IM'99), pp. 611-622, May 1999.

[FAR96] Farmer W., Guttman J., and Swarup V., "Security for mobile agents: Issues and requirements", Proc. of the 19th National Information Systems Security Conf., pp. 591-597, Oct. 1996.

[FER01] Feridun M., Krause J., "A Framework for Distributed Management with Mobile Components", Computer Networks, 35(1), pp. 25-38, Jan. 2001.

[FIPA] Foundation for Intelligent Physical Agents (FIPA), http://www.fipa.org/.

[FUG98] Fuggetta A., Picco G.P., Vigna G., "Understanding Code Mobility", IEEE Transactions on Software Engineering, 24(5), pp. 342-361, 1998.

[FUN98] Fünfrocken S., "Transparent Migration of Java-Based Mobile Agents: Capturing and Re-establishing the State of Java Programs", Proc. of the 2nd Int. Workshop on Mobile Agents (MA'98), LNCS vol. 1477, pp. 26-37, Oct. 1998.

[GAV99] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "A Progressive Network Management Architecture Enabled by Java Technology", Proc. of the 1st Int. Conf. and Exhibition on the Practical Applications of Java (PA-Java'99), pp. 61-72, April 1999.

[GHA99] Ghanea-Hercock R., Collis J.C., Ndumu D.T., "Co-operating Mobile Agents for Distributed Parallel Processing", Proc. of the 3rd ACM Conf. on Autonomous Agents (Agents'99), pp. 398-399, May 1999.

[GOK98] Gokhale A., Schmidt D., “Evaluating CORBA Latency and Scalability Over High-Speed Networks”, IEEE Transactions on Computers Journal, April 1998.

[GOL91] Goldszmidt G., Yemini Y., Yemini S., “Network Management by Delegation”, Proc. of the 2nd Int. Symposium on Integrated Network Management (ISINM’91), April 1991.

[GOL93a] Goldszmidt G., Yemini Y., “Evaluating Management Decisions via Delagation”, Proc. of the 3rd Int. Symposium on Integrated Network Management (ISINM’93), April 1993.

[GOL93b] Goldszmidt G., “On Distributed Systems Management”, Proc. of the 3rd IBM/CAS Conf., Oct. 1993.

[GOL95] Goldszmidt G., “Distributed Management by Delegation”, PhD thesis, Columbia University, New York, Dec. 1995.

[GOL96] Goldszmidt G., “Network Management Views using Delegated Agents”, Proc. of the 6th IBM/CAS Conf., 1996.

[GOL98] Goldszmidt G., Yemini Y., “Delegated Agents for Network Management”, IEEE Communications Magazine, Special Edition on Network Management Paradigms, March 1998.

[GRA95] Gray R., “Agent Tcl: A Transportable Agent System”, Proc. of the CIKM Workshop on Intelligent Information Agents, 4th Int. Conf. on Information and Knowledge Management, 1995.

[Grasshopper] IKV++ GmbH - Grasshopper, http://www.ikv.de/products/grasshopper/.

[GRE97] Green S., Hurst L., Nagle B., Cunningham P., Somers F., Evans R., “Software Agents: A review”, Technical report, Department of Computer Science, Trinity College, Dublin, Ireland, May 1997.

[GRE99] Greenwood D., Gavalas D., “Using Active Processes as the Basis for an Integrated Distributed Network Management Architecture”, Proc. of the 1st Int. Working Conf. on Active Networks (IWAN’99), LNCS vol. 1653, pp. 199-211, July 1999.

[GRI97] Grimes G., “Intelligent Agents for Network Fault Diagnosis and Testing”, Proc. of the 5th Int. Symposium on Integrated Network Management (IM’97), pp. 232-244, May 1997.

[GSC99] Gschwind T., Feridun M., Pleisch S., “ADK - Building Mobile Agents for Network and Systems Management from Reusable Components”, Proc. of the

Joint Symposium: 1st Int. Symposium on Agent Systems and Applications / 3rd Int. Symposium on Mobile Agents (ASA/MA'1999), Sep. 1999.

[HAR95] Harrison C., Chess D., Kershenbaum A., "Mobile Agents: Are they a good Idea?", IBM Research Report, RC 19887, March 1995, http://www.infosys.tuwien.ac.at/Research/Agents/archive/mobagtibm.ps.gz .

[HAR99] Harrington D., Presuhn R., Wijnen B., "An Architecture for Describing SNMP Management Frameworks", RFC 2571, April 1999.

[HAY99] Hayzelden A.,Bigham J., "Agent Technology in Communications Systems: An Overview", Knowledge Engineering Review, 14(4), pp. 341-375, Dec. 1999.

[HEG94] Hegering H.G., Abeck S., "Integrated Network and System Management", Addison-Wesley, 1994.

[HUN97] Hunt R., "SNMP, SNMPv2 and CMIP – The Technologies for Multivendor Network Management", Computer Communications, 20(2), pp. 73-78, March 1997.

[INCO] EU-Inco 977113 Project, Gulyas L., Kovacs L., Micsik A., Pataki B., Zsamboki I., "Mobile Software Agents", public deliverable, June 2000.

[ISM99] Ismail L., Hagimont D., Mossiere J., "Evaluation of the Mobile Agent Technology: Comparison with the Client/Server Paradigm", Proc. of the Int. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99), pp. 306-313, Nov. 1999.

[ISO91a] Information Technology, Open Systems Interconnection, "Systems Management Overview", Sep. 1991.

[ISO91b] Information Technology, Open Systems Interconnection, "Common Management Information Protocol (CMIP) – Part 1: Specification", ISO/IEC 9596, Nov. 1991.

[ISO92] ITU-T Rec. X.739/X.738, Information Technology - Open Systems Interconnection, "Systems Management - Part 13: Summarization Function", DIS 10164-13, ISO/IEC, Oct. 1992.

[ISO93] ITU-T Rec. X.739/X.738, Information Technology - Open Systems Interconnection, "Systems Management - Part 11: Metric Objects and Attributes", DIS 10164-11, ISO/IEC, Oct. 1993.

[ITU94] ITU-T, Recommendation X.690. Information Technology, "ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)". ITU, July 1994.

[JAMES] JAMES: A Mobile Agent Platform for the Management of Telecommunication and Data Networks, University of Coimbra, Portugal, SIEMENS Portugal, http://james.dei.uc.pt/.

[JAR] Sommerer A., Sun Microsystems, “The Java Tutorial: JAR files”, http://java.sun.com/docs/books/tutorial/jar/.

[Jasmin] “The Jasmin Project, A Script-MIB Implementation”, http://www.ibr.cs.tu-bs.de/projects/jasmin/.

[JAVA] Sun Microsystems, “Java Language Overview”, White paper, http://java.sun.com/docs/overviews/java/java-overview-1.html.

[JB] Sun Microsystems, JavaBeans specification, Dec. 1996, http://java.sun.com/beans/docs/spec.html

[JBM] The Java-Based Management Page, http://joe.lindsay.net/javamgmt.html.

[JCA] Sun Microsystems, “Java Cryptography Architecture API Specification & Reference”, Dec. 1999, http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html.

[JDBC] Sun Microsystems, JDBC Database Access tutorial, http://java.sun.com/docs/books/tutorial/jdbc/index.html.

[JDMK] Sun Microsystems, Java Dynamic Management Kit, (JDMK), 1998, http://www.sun.com/software/java-dynamic/.

[JEN00] Jennings N.R., “On Agent-Based Software Engineering”, Artificial Intelligence, 117(2), pp. 277-296, 2000.

[JIA00] Jiao J., Naqvi S., Raz D., Sugla B., “Towards Efficient Monitoring”, IEEE Journal of Selected Areas in Communications, 5(18), pp. 723-732, May 2000.

[JIT] Java JIT Compiler Overview, http://www.sun.com/solaris/jit/.

[JMAPI] Sun Microsystems, Java Management API Architecture, Revision A, Sep. 1996.

[JMX] Sun Microsystems, Java Management Extensions (JMX) specification, 1999, http://java.sun.com/products/JavaManagement/index.html.

[JNI] Sun Microsystems, Java Native Interface (JNI) tutorial, http://java.sun.com/docs/books/tutorial/native1.1/index.html.

[JSAPI] Sun Microsystems, Java Security API Overview, Feb. 1997, http://www.javasoft.com/products/jdk/1.1/docs/guide/security/JavaSecurityOverview.html.

[KAH97] Kahani M., Beadle H., “Decentralised Approaches for Network Management”, Computer Communications Review, 27(3), pp. 36-47, July 1997.

[KAL97] Kalyanasundaram P., Sethi S., Sherwin M., Zhu D., “A Spreadsheet-Based Scripting Environment for SNMP”, Proc. of the 5th IFIP/IEEE Int. Symposium on Integrated Network Management (IM’97), pp. 752-765, May 1997.

[KAR98a] Karygiannis A., “Network Security Testing Using Mobile Agents”, Proc. of the Int. Conf. on Telecommunications (ICT’98), July 1998.

[KAR98b] Karjoth G., Asokan N., Gülcü C., “Protecting the Computation Results of Free-Roaming Agents”, Proc. of the 2nd Int. Workshop on Mobile Agents (MA’98), LNCS vol. 1477, pp. 195–207, Sep. 1998.

[KAW00] Kawamura R., Stadler R., “Active Distributed Management for IP Networks”, IEEE Communications Magazine, 38(4), pp. 114-120, April 2000.

[KEL99] Keller A., Reiser, H., “Dynamic Management of Internet Telephony Servers: A Case Study based on JavaBeans and JDMK”, Proc. of the 3rd Int. Enterprise Distributed Object Computing Conf. (EDOC’99), pp. 135 - 146, Sep. 1999.

[KER93] Kershenbaum A., “Telecommunications Network Design Algorithms”, McGraw-Hill eds., 1993.

[KIM98] Kim G.J., Kim Y.S., Lee H.H., “A Design of Management System for ATM Switches Using Mobile Agent Concept”, Proc. of the IEEE Global Telecommunications Conf. (Globecom’98), pp. 2504-2508, Nov. 1998.

[KNI99] Knight G., Hazemi R., “Mobile Agent based Management in the INSERT Project”, Journal of Network and Systems Management, 7(3), Sep. 1999.

[KOO95] Kooijman R., “Divide and Conquer in Network Management Using Event-Driven Network Area Agents”, Technical University of Delft, The Netherlands, May 1995.

[KOT99] Kotz D., Gray R., “Mobile Agents and the Future of the Internet”, ACM Operating Systems Review, 33(3), pp. 7-13, August 1999.

[KQML] Finin T., Fritzson R., McKay D., McEntire R., “KQML as an Agent Communication Language”, Proc.of the 3rd Int. Conf. on Information and Knowledge Management (CIKM’94), pp. 456-463, Nov. 1994.

[KRA98] Krause J., “Technology Review of Java-Based Mobile Agent Platforms”, Technical Report SSC/1998/009, SSC, EPFL, Lausanne, Switzerland, Feb. 1998, http://sscwww.epfl.ch/Pages/publications/ps_files/tr98_010.ps.

[KU97] Ku H., Luderer G., Subbiah B., "An Intelligent Mobile Agent Framework for Distributed Network Management", Proc. of the IEEE Global Telecommunications Conf. (Globecom ’97), pp. 160-164, Nov. 1997.

[LAN98] Lange D., Oshima M., “Programming and Deploying Java Mobile Agents with Aglets”, Addison-Weslay, 1998.

[LAN99] Lange D., Oshima M., “Seven Good Reasons for Mobile Agents”, Communications of the ACM, 42(3), pp. 88-89, March 1999.

[LAZ97] Lazar S., Sidhu D., "Discovery: A Mobile Agent Framework for Distributed Application Development", Technical Report, Maryland Center for Telecommunications Research, University of Maryland, 1997.

[LAZ98] Lazar S., Weerakoon I., Sidhu D., “A Scalable Location Tracking and Message Delivery Scheme for Mobile Agents”, Proc. of the 7th IEEE WETICE Conf., June, 1998.

[LEE00] Lee J-O, “Enabling Network Management Using Java Technologies”, IEEE Communications Magazine, 38(1), pp.116-123, Jan. 2000.

[LEV99] Levi D., Schoenwaelder J., “Definitions of Managed Objects for the Delegation of Management Scripts”, RFC 2592, May 1999.

[LIN99] Lin H.C., Lai H.L., Lai S.C., “Automatic Link Layer Topology Discovery of IP Networks”, Proc. of the IEEE Int. Conf. on Communications (ICC’99), June 1999.

[LIO98] Liotta A., Knight G., Pavlou G., “Modelling Network and System Monitoring Over the Internet with Mobile Agents”, Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS’98), pp. 303-312, Feb. 1998.

[LIO99] Liotta A., Knight G., Pavlou G., “On the Performance and Scalability of Decentralised Monitoring Using Mobile Agents”, Proc. of the 10th IFIP/IEEE Int. Workshop on Distributed Systems: Operations & Management (DSOM’99), pp. 3-18, Oct. 1999.

[LIO01] Liotta A., Pavlou G., Knight G., “Active Distributed Monitoring”, Proc. of the 2001 Int. Conf. on Communications (ICC’01), in press, June 2001.

[LIP00] Lipperts S., “How to Efficiently Deploy Mobile Agents for an Integrated Management”, Proc. of the 3rd IFIP/GI Int. Conf. on Trends towards a Universal Services Market (USM’2000), Sep. 2000.

[LOP00] Lopes R.P., Oliveira J.L., “On the Use of Mobility in Distributed Network Management”, Proc. of the 33rd Hawaii International Conference on System Sciences (HICSS-33), Jan. 2000.

[LUD97] Luderer G., Ku H., Subbiah B., Narayanan A., “Network Management Agents supported by a Java environment”, Proc. of the 5th IFIP/IEEE Int. Symposium on Integrated Network Management (IM’97), May 1997.

[MAG96] Magedanz T., Rothermel K., Krause S., “Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?”, Proc. of the IEEE Conf. on Computer Communications (Infocom’96), March 1996.

[MAL] Mobile Agent List, http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html.

[MAP] MAP (Mobile Agent Platform) project, Università di Catania (Italy), Università di Messina, Italy, http://sun195.iit.unict.it/MAP/.

[MAR98] Martin-Flatin J.P., “IP Network Management Platforms Before the Web”, Technical Report SSC/1998/021, SSC, EPFL, Lausanne, Switzerland, Dec. 1998, http://icawww2.epfl.ch/~jpmf/papers/tr_1998_021.pdf.

[MAR99a] Martin-Flatin J.P., Znaty S., Hubaux J.P., “A Survey of Distributed Enterprise Network and Systems Management Paradigms”, Journal of Network and Systems Management, 7(1), pp. 9-26, March 1999.

[MAR99b] Martin-Flatin J.P., “Push vs. Pull in Web-Based Network Management”, Proc. of the 6th IFIP/IEEE Int. Symposium on Integrated Network Management (IM’99), pp. 3-18, May 1999.

[MAR00] Martin-Flatin J.P., “Web-Based Management of IP Networks and Systems“, Ph.D. thesis, Swiss Federal Institute of Technology, Lausanne (EPFL), Oct. 2000.

[MAR01] Marques P., Simões P., Silva L.M., Boavida F., Silva J.G., “Providing Applications With Mobile Agent Technology”, 4th IEEE Int. Conf. on Open Architectures and Network Programming (Openarch’01), in press, April 2001.

[MARINE] AC340 MARINE (Mobile Agent enviRonments in Intelligent NEtworks) project, “MARINE Agent Platforms”, public deliverable I1201, Sep. 1998.

[MASIF] GMD Fokus, IBM Corp., “The OMG Mobile Agent System Interoperability Facility (MASIF) Standard”, http://www.fokus.gmd.de/research/cc/ima/masif/.

[MAZ96] Mazumdar, S., “Inter-Domain Management between CORBA and SNMP: Web-based Management - CORBA/SNMP Gateway Approach”, Proc. of the 7th

IFIP/IEEE Int. Workshop on Distributed Systems: Operations & Management (DSOM'96), Oct. 1996.

[McC91] McCloghrie K., Rose M., "Management Information Base for Network Management of TCP/IP-based Internets: MIB-II", RFC 1213, March 1991.

[McC96] McCloghrie K., "An Administrative Infrastructure for SNMPv2", RFC 1909, Feb 1996.

[McK00] McKee P., Marshall I., Henning I., "Research Directions in Distributed Systems", BT Technology Journal, 18(2), April 2000.

[McM99] McManus E., "Script MIB Implementation Experience", Simple Times, 7(2), Nov. 1999, http://www.simple-times.org/pub/simple-times/issues/7-2.html.

[MEY95] Meyer K., Erlinger M., Betser J., Sunshine C., Goldszmidt G., Yemini Y., "Decentralizing Control and Intelligence in Network Management", Proc. of the 4th Int. Symposium on Integrated Network Management (ISINM'95), pp.4-16, May 1995.

[MIAMI] MIAMI: Mobile Intelligent Agents for Managing the Information Infrastructure, http://www.fokus.gmd.de/research/cc/ecco/miami/.

[MIAMI98] MIAMI project public deliverable, "Platform Enhancement Requirements", Document number: AC338-GMD-WP2-DS-016, Aug. 1998, available at: http://www.fokus.gmd.de/research/cc/ecco/miami/public/doc/entry.html.

[MIL99] Milojicic D., "Mobile agent applications", Trend Wars, IEEE Concurrency, 7(3), July-Sep. 1999, http://computer.org/concurrency/pd1999/pdf/p3080.pdf.

[MIN99] Minar N., Kramer K.H., Maes P., "Cooperating Mobile Agents for Dynamic Network Routing", in "Software Agents for Future Communications Systems", Hayzelden A. and Bigham J. (Eds.), Springer-Verlag, 1999.

[MIT94] Mitra N., "Efficient Encoding Rules for ASN.1-Based Protocols", AT&T Technical Journal, 73(3), pp. 80-93, 1994.

[MLM] SNMP research, "The Mid-Level Manager", http://www.snmp.com/products/mlm.html.

[MOR97] Morgan B., "CORBA meets Java - Distributed Object alternatives: DCOM and RMI", JavaWorld, 2(10), Oct. 1997, http://www.javaworld.com/javaworld/jw-10-1997/.

[MOU98a] Mountzia M-A, "A Distributed Management Approach Based on Flexible Agents", Interoperable Communication Networks, 1(1), pp. 99-120, 1998.

[MOU98b] Mountzia M-A, "Flexible Agents in Integrated Network and Systems Management", PhD thesis, Technische Universität München, 1998.

[MOU99] Mountzia M-A., Rodosek G.D., "Using the Concept of Intelligent Agents in Fault Management of Distributed Services", Journal of Network and Systems Management, 7(4), Dec. 1999.

[MUL98] Müller, J.P., "Architectures and applications of intelligent agents: A survey", The Knowledge Engineering Review, 13(4), pp. 353-80, 1998.

[NIC98] Nicklisch J., Quittek J., Kind A., Arao S., "INCA: An Agent-Based Network Control Architecture", Proc. of the 2nd Int. Workshop on Intelligent Agents for Telecommunication Applications (IATA'98), LNCS vol. 1437, pp. 143-155, July 1998.

[NWA96] Nwana H., Ndumu D., "Introduction to Agent Technology", BT Technology Journal, 14(4), pp. 55-67, 1996.

[OLI99] Oliveira J.L., Lopes R.P., "Distributed Management Based on Mobile Agents", Proc. of the 1st Int. Workshop on Mobile Agents For Telecommunication Applications (MATA'99), pp. 243-258, Oct. 1999.

[OSS97] Sun Microsystems, Object Serialisation Specification, 1997, http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serial-title.doc.html.

[PAG98] Pagurek B., Li Y., Bieszczad A., Susilo G., "Network Configuration Management in Heterogeneous ATM Environments", Proc. of the 2nd Int. Workshop on Agents in Telecommunications Applications (IATA'98), Paris, France, July 1998.

[PAG00] Pagurek B., Wang Y., White T., "Integration of Mobile Agents with SNMP: Why and How", Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS'2000), pp. 609-622, April 2000.

[PAP99] Papastavrou S., Samaras G., Pitoura E., "Mobile Agents for WWW Distributed Database Access", Proc. of the 15th Int. Conf. on Data Engineering (ICDE'99), March 1999.

[PAR98] Park J.K., Ban N.J., Kim T.G., "Java-Based Network Management Environment", Proc. of the IEEE Int. Conf. on Communications (ICC'98), pp. 1124-1128, June 1998.

[PAU98] Paul H., Krishnan S., "The benefits of CORBA-based network management", Communications of the ACM, 41(10), p.73-79, Oct. 1998.

[PAV96] Pavlou G., Mykoniatis G., Sanchez J., "Distributed Intelligent Monitoring and Reporting Facilities", IEE Distributed Systems Engineering Journal, (3)2, pp. 124-135, 1996.

[PAV00] Pavlou G., "Using Distributed Object Technologies in Telecommunications Network Management", IEEE Journal of Selected Areas in Communications, 18(5), pp. 644-653, May 2000.

[PAV01] Pavlou G., "Network & Service Management and Control", Notes for the MSc in Communication Networks and Software, University of Surrey, 2001.

[PEI97] Peine H., Stolpmann T., "The Architecture of the Ara Platform for Mobile Agents", Proc. of the First Int. Workshop on Mobile Agents (MA'97), April 1997.

[PER97] Perkins D.T., "SNMP Versions". The Simple Times, 5(1), pp. 13–14, Dec. 1997, http://www.simple-times.org/pub/simple-times/issues/5-1.html.

[PHA98] Pham V., Karmouch A., "Mobile Software Agents: An Overview", IEEE Communications Magazine, 36(7), pp. 26-37, 1998.

[PIC98] Picco G.P., "Understanding, Evaluating, Formalizing and Exploiting Code Mobility", PhD thesis, Politecnico di Torino, Feb. 1998.

[PIC01] Picco G.P., "Mobile Agents: An Introduction", Microprocessors and Microsystems, in press, 2001.

[picoJava] Sun Microsystems, picoJava technology, http://www.sun.com/microelectronics/communitysource/picojava/.

[PIN99] Pinheiro R., Poylisher A., Caldwell H., "Mobile Agents for Aggregation of Network Management Data", Proc. of the Joint Symposium: 1st Int. Symposium on Agent Systems and Applications / 3rd Int. Symposium on Mobile Agents (ASA/MA'99), Sep. 1999.

[PMP] Perpetuum Mobile Procura Project, Network Management and Artificial Intelligence Laboratory, Department of Systems and Computer Engineering, Carleton University, Canada, http://www.sce.carleton.ca/netmanage/perpetuum.shtml.

[Providers] "How to Implement a Provider for the Java Cryptography Architecture", Sep. 1998, http://java.sun.com/products/jdk/1.2/docs/guide/security/HowToImplAProvider.html.

[PUL99] Puliafito A., Riccobene S., Scarpa M., "An analytical comparison of the client-server, remote evaluation and mobile agents paradigms", Proc. of the Joint Symposium: 1st Int. Symposium on Agent Systems and Applications / 3rd Int. Symposium on Mobile Agents (ASA/MA'1999), Sep. 1999.

[PUL00a] Puliafito A., Tomarchio O., Vita L., "MAP: Design and Implementation of a Mobile Agents Platform", Journal of System Architecture, 46(2), pp.145-162, Jan. 2000.

[PUL00b] Puliafito A. Tomarchio O., "Using Mobile Agents to implement flexible Network Management strategies", Computer Communications, 23(8), pp. 708-719, April 2000.

[QUE97] Quendt B., "An Agent-Based Resource-Control in a Signalling System for the Open Telecommunication Market", Proc. of the 2nd Int. Conf. and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents (PAAM'97), pp. 381-394, April 1997.

[RAZ00] Raz D., Shavitt Y., "Active Networks for Efficient Distributed Network Management", IEEE Communications Magazine, 38(3), pp. 138-143, March 2000.

[RES97] Rescigno A., "Optimal Polling in Communication Networks", IEEE Transactions on Parallel and Distributed Systems, 8(5), 1997.

[RFCs] Network Management RFCs, http://www.iol.unh.edu/consortiums/netmgt/rfc-main.html.

[RIV78] Rivest R.L., Shamir A., Adleman L., "A Method for obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, 21(2), Feb. 1978.

[RMI] Sun Microsystems, Java Remote Method Invocation (RMI), http://java.sun.com/products/jdk/rmi/ index.html.

[ROS90] Rose M., "Bulk Table Retrieval with the SNMP", RFC 1187, Oct. 1990.

[ROS96] Rose M., "The Simple Book: an Introduction to Networking Management", 2nd Ed., Prentice Hall, 1996.

[ROT97] Rothermel K., Hohl F., Radouniklis N., "Mobile Agent Systems: What is Missing?", Proc. of the 1st IFIP Int. Working Conf. on Distributed Applications and Interoperable Systems (DAIS'97), pp. 111-124, Sep. 1997.

[RUB99] Rubinstein M., Duarte O.C., “Evaluating Tradeoffs of Mobile Agents in Network Management”, Networking and Information Systems Journal, 2(2), pp. 237-252, July 1999.

[RUB00] Rubinstein, M., Duarte O. C., “Using Mobile Agent Strategies for Reducing the Response Time in Network Management”, Proc. of the 16th IFIP World Computer Congress (ICCT’2000), pp. 278-281, Aug. 2000.

[SAH97] Sahai A., Billiart S., Morin C., “Astrolog: A Distributed and Dynamic Environment for Network and System Management”, Proc. of the 1st European Information Infrastructure User Conf., Feb. 1997.

[SAH98] Sahai A., Morin C., “Enabling a Mobile Network Manager through Mobile Agents”, Proc. of the In Proc. of the 2nd Int. Workshop on Mobile Agents (MA’98), LNCS vol. 1477, pp. 249-260, Sep. 1998.

[SAN98] Sander T., Tschudin C.,. “Protecting Mobile Agents Against Malicious Hosts”. In Vigna G. (Ed.). Mobile Agents and Security, LNCS vol. 1419, pp. 44–60, Springer, 1998.

[SCH97] Schoonderwoerd R., Holland O., Bruten J., “Ant-like agents for load balancing in telecommunications networks”, Proc. of the 1st Int. Conf. on Autonomous Agents (Agents’97), Feb. 1997.

[SCH98] Schramm C., Bieszczad A., Pagurek B., “Application-Oriented Network Modelling with Mobile Agents”, Proc. of the IEEE/IFIP Network Operations and Management Symposium NOMS’98, Feb. 1998.

[SCH00] Schönwälder J., Quittek J., Kappler C., “Building Distributed Management Applications with the IETF Script MIB”, IEEE Journal on Selected Areas in Communications, 18(5), pp.702-714, May 2000.

[SIE96] Siegl M. R., and Trausmuth G., “Hierarchical Network Management: A Concept and its Prototype in SNMPv2”, Computer Networks and ISDN Systems, 28(4), pp. 441-452, Feb. 1996.

[SIL99a] Silva L.M., Simoes P., Soares G., Martins P., Batista V., Renato C., Almeida L., Stohr N., “JAMES: A Platform of Mobile Agents for the Management of Telecommunication Networks”, Proc. of the 3rd Int. Workshop on Intelligent Agents for Telecommunication Applications (IATA’99), LNCS vol. 1699, pp. 76-95, Aug. 1999.

[SIL99b] Silva L.M., Batista V., Martins P., Soares G., “Using Mobile Agents for Parallel Processing”, Proc. of the Int. Symposium on Distributed Objects and Applications (DOA’99), Sep. 1999.

[SIL00] Silva L., Soares G., Martins P., Batista V., Santos L., “Comparing the Performance of Mobile Agent Systems: A Study of Benchmarking”, Computer Communications, 23(8), pp. 769-778, April 2000.

[SIM99] Simoes P., Silva L.M., Fernandes F.B., “Integrating SNMP into a Mobile Agent Infrastructure”, Proc. of the 10th IFIP/IEEE Int. Workshop on Distributed Systems: Operations & Management (DSOM’99), Oct. 1999.

[SLO94] Sloman M. (eds.), “Network and Distributed Systems Management”, Addison-Wesley, 1994.

[SOA99] Soares G., Silva L.M., “Optimizing the Migration of Mobile Agents”, Proc. of the 1st Int. Workshop on Mobile Agents for Telecommunication Applications (MATA’99), pp. 161-178, Oct. 1999.

[SOMA] SOMA: Secure and Open Mobile Agent, Laboratory of Advanced Research on Computer Science, Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Italy, http://www-lia.deis.unibo.it/Software/SOMA/.

[SPR99] Sprenkels R., Martin-Flatin J.P., “Bulk Transfers of MIB Data”, The Simple Times, 7(1), pp. 1-7, March 1999, http://www.simple-times.org/pub/simple-times/issues/7-1.html.

[STA90] Stamos J.W., Grifford D.K., “Implementing Remote Evaluation”, IEEE Transactions on Software Engineering, 16(7), pp. 710-22, July 1990.

[STA98] Stallings W., “SNMPv3: A Security Enhancement to SNMP”. IEEE Communications Surveys, 1(1), pp. 2–17, March 1998.

[STA99] Stallings W., “SNMP, SNMPv2, SNMPv3 and RMON 1 and 2”, 3rd ed., Addison Wesley, 1999.

[STR96] Straßer M., Baumann J., Hohl, F., “Mole-A Java Based Mobile Agent System”, Proc. of the Second European Conf. on Object Oriented Programming (ECOOP’96), p327-334, July 1996.

[STR97] Straßer M., Schwehm M., “A Performance Model for Mobile Agent Systems”, Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA’97), pp. 1132-1140, June 1997.

[STR99] Strassner J., Baker F., “Directory Enabled Networking”, Macmillan Technical Publishing, 1999.

[SUG99] Sugauchi K., Miyazaki S., Covaci S., Zhang T., “Efficiency Evaluation of a Mobile Agent Based Network Management System”, 6th Int. Conf. on

Intelligence and Services in Networks (IS&N'99), LNCS vol. 1597, pp. 527-535, April 1999.

[SunTools] "Sun.tools", http://www.cis.ohio-state.edu/~gb/cis788.07r/javac/packages.html.

[SUR00] Suri N., Bradshaw J., Breedy M., Groth P., Hill G., Jeffers R., "Strong Mobility and Fine-Grained Resource Control in NOMADS", Proc. of the Joint Symposium: 2nd Int. Symposium on Agent Systems and Applications / 4th Int. Symposium on Mobile Agents (ASA/MA'2000), Sep. 2000.

[SUS98] Susilo G., Bieszczad A., Pagurek B., "Infrastructure for Advanced Network Management based on Mobile Code", Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS'98), pp. 322-333, Feb. 1998.

[TAC00] Tacker M., "Java I/O and Compression", ACM Crossroads, No 4, March 2000, http://www.acm.org/crossroads/xrds6-3/ovp63.html.

[TAN96] Tanenbaum A., "Computer Networks", Prentice-Hall, 3rd ed., 1996.

[TEN96] Tennenhouse D., Wetherall D., "Towards an Active Network Architecture", ACM Computer Communication Review, 26(2), pp. 5-18, April 1996.

[TINA93] Rubin H., "An Overview of the TINA Consortium Work", Document TB-G1.HR.001-1.0-93, TINA Consortium, Dec. 1993.

[UCD-SNMP] "The UCD-SNMP Home Page", http://ucd-snmp.ucdavis.edu/.

[VAS97] Vassila N., Pavlou G., Knight G., "Active Objects in TMN", Proc. of the IFIP/IEEE Symposium on Integrated Network Management (IM'97), pp. 139-150, May 1997.

[VEN98] Venners B., "Designing with Dynamic Extension: How Dynamic Extension Works in Java and How to Use it in Your Designs", Dec. 1998, http://www.artima.com/designtechniques/dynaext.html.

[VIG98] Vigna G. (Eds.). Mobile Agents and Security. LNCS vol. 1419, Springer, 1998.

[Voyager] ObjectSpace, Voyager, http://www.objectspace.com/products/voyager/.

[WAD95] Waldbusser S., "Remote Network Monitoring Management Information Base", RFC 1757, Feb. 1995.

[WBEM] Distributed Management Task Force, WBEM Initiative, Feb. 2000 http://www.dmtf.org/wbem/.

[Web] The Web Management Page, http://joe.lindsay.net/webbased.html.

[WEL96] Wellens C., Auerbach K., "Towards Useful Management", Simple Times, 4(3), pp. 1- 6, July 1996, http://www.simple-times.org/pub/simple-times/issues/4-3.html.

[WHI96] White J.E., "Telescript Technology: Mobile Agents", White paper, General Magic Inc., 1996.

[WHI98] White T., Bieszczad A., Pagurek B., "Distributed Fault Location in Networks Using Mobile Agents", Proc. of the 2nd Int. Workshop on Intelligent Agents for Telecommunication Applications (IATA'98), LNCS vol. 1437, pp. 130-141, July 1998.

[WHI99] White T., Pagurek B., Bieszczad A., "Network Modelling for Management Applications Using Intelligent Mobile Agents", Journal of Network and Systems Management, 7(3), pp. 295-321, Sep. 1999.

[Win32] Platfptm SDK: Win32 API, http://msdn.microsoft.com/library/psdk/portals/win32start_1n6t.htm.

[Win32Enum]Microsoft support, "How to Enumerate Applications in Win32", http://support.microsoft.com/support/kb/articles/Q175/0/30.asp.

[Windump] "WinDump: tcpdump for Windows", Computer Network Group, Politecnico di Torino, http://netgroup-serv.polito.it/windump/.

[YUC00] Yucel S., Saydam T., "Distributed Connection Management Using Mobile Agent Technology", Proc. of the 2000 Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), June 2000.

[ZAP97] Zapf M., "Design Paradigms in Agent-Based Systems", Proc. of the 1st IFIP Int. Working Conf. on Distributed Applications and Interoperable Systems (DAIS'97), pp. 101-107, Sep. 1997.

[ZAP99] Zapf M., Herrmann K., Geihs K., "Decentralized SNMP Management with Mobile Agents", Proc. of the 6th IFIP/IEEE Int. Symposium on Integrated Network Management (IM'99), pp. 623-635, May 1999.

[ZHA98a] Zhang T., Magedanz T., Covaci S., "Mobile Agents vs. Intelligent Agents - Interoperability and Integration Issues", 4th Int. Symposium on Interworking, July 1998.

[ZHA98b] Zhang D., Zorn W., "Developing Network Management Applications in an Application-Oriented Way Using Mobile Agent", Computer Networks And ISDN Systems, 30(16-18), pp. 1551-1557, Sep. 1998.

APPENDIX A

LIST OF PUBLICATIONS

- **Journals:**

[GAV00a] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "Advanced Network Monitoring Applications Based on Mobile/Intelligent Agent Technology", *Computer Communications*, Elsevier Science, special issue on "Mobile Software Agents for Telecommunication Applications", Vol. 23, No 8, pp. 720-730, April 2000. The paper was selected as the best paper published in 1999-2000 by a postgraduate student within the Electronic Systems Engineering Department, University of Essex, and received the "British Telecom Postgraduate Publication Prize".

[GAV01a] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "Mobile Software Agents for Decentralised Network & Systems Management", *invited paper*, to appear in *Microprocessors and Microsystems*, Elsevier Science, special issue on "Mobile Agent Technology: from first proposals to current evolutions".

[GAV01b] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "Hierarchical Network Management: A Scalable and Dynamic Mobile Agent-Based Approach", *submitted* to *Computer Networks*, Elsevier Science.

- **Refereed Conferences, Symposiums and Workshops:**

[GAV99a] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "A Progressive Network Management Architecture Enabled by Java Technology", Proceedings of the *1st International Conference and Exhibition on the Practical Applications of Java* (*PA-Java'99*), pp. 61-72, London, UK, 21-23 April 1999.

[GAV99b] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "An Infrastructure for Distributed and Dynamic Network Management based on Mobile Agent Technology", Proceedings of the *IEEE International Conference on Communications* (*ICC'99*), Vol. 2, pp. 1362-1366, Vancouver, Canada, 6-10 June 1999.

[GRE99] Greenwood D., Gavalas D., "Using Active Processes as the Basis for an Integrated Distributed Network Management Architecture", Proceedings of the *1st International Working Conference on Active Networks* (*IWAN'99*), LNCS Vol. 1653, Springer-Verlag, Covaci S. (Eds.), pp. 199-211, Berlin, Germany, 30 June - 2 July 1999.

[GAV99c] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "A Hybrid Centralised - Distributed Network Management Architecture", Proceedings of the *4th IEEE International Symposium on Computers and Communications* (*ISCC'99*), pp. 434-441, Sharm El Sheikh, Egypt, 6–8 July 1999.

[GAV99d] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "Using Mobile Agents for Distributed Network Performance Management", Proceedings of the *3rd International Workshop on Intelligent Agents for Telecommunication Applications* (*IATA'99*), LNCS Vol. 1699, Springer-Verlag, Albayrak, S. (Eds.), pp. 96-112, Stockholm, Sweden, 9-11 August 1999.

[GAV99e] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "Complementary Polling Modes for Network Performance Management Employing Mobile Agents", Proceedings of the *IEEE Global Communications Conference* (*Globecom'99*), pp. 401-405, Rio de Janeiro, Brazil, 5-9 December 1999.

[GAV00b] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "Deploying a Hierarchical Management Framework Using Mobile Agent Technology", Proceedings of the *7th International Conference on Intelligence in Services and Networks* (*IS&N'2000*), LNCS Vol. 1774, Springer-Verlag, Delgado J., Stamoulis G.D., Mullery A., Prevedourou D. and Start K. (Eds.), pp. 333-348, Athens, Greece, 23-25 February 2000.

[GAV00c] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "Enabling Mobile Agent Technology for Intelligent Bulk Management Data Filtering", Proceedings of the *2000 IEEE/IFIP Network Operations and Management Symposium* (*NOMS'2000*), pp.623-636, Honolulu, Hawaii, U.S.A., 10-14 April 2000.

[GAV00d] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., "Implementing a Highly Scalable and Adaptive Agent-Based Management Framework", Proceedings of the *IEEE Global Communications Conference* (*Globecom'2000*), Vol. 3, pp. 1458-1462, San Francisco, U.S.A., 27 November-1 December 2000.

APPENDIX B

PLATFORM DESIGN OPTIMISATIONS

B.1. INTRODUCTION

The framework described in Chapter 4 represents a Mobile Agent Platform (MAP) with the core functionality needed for Network & Systems Management (NSM) applications. Several design decisions are incorporated aiming at minimising the use of network resources. Most of these efforts have been directed towards diminishing the effect of Mobile Agent (MA) code transfers, acknowledging the fact that the bytecode size is typically far larger than the corresponding state size. Therefore, through the proposed bytecode distribution scheme (see Section 4.4.1.4.1), MA code transfers are performed only once, at the MAs creation time; following that, only MA state transfers are involved in every migration. Although the gain achieved over publicly available MAPs, which enforce the transfer of both the state and code in every migration, is already evident, it is still important to reduce the state size as much as possible. This issue is investigated in Section B.2, with Section B.3 looking at programming techniques that minimise MAs migration latency.

B.2. MINIMISING MOBILE AGENT STATE SIZE

MAs own their mobility characteristic to their ability to carry their persistent state as the latter represents the knowledge/data collected throughout their lifecycle. The state comprises the output of the *serialisation* process (see Section 3.5.2). At the destination end, the state is reconstructed through the inverse process of *de-serialisation*. MAs state includes information about the MA class (e.g. the name of the class, the package that this class belongs to, etc.) and the values of the non-transient[1] and non-static variables/objects declared within the MA class [OSS97]. Should this class extends another class defining the basic MA functionality, the MA's state will include the values of the transient objects declared within the superclass as well.

Since the proposed framework involves *only* the transfer of MA objects state and not their code, it is apparent that the minimisation of the state size is a crucial factor on reducing the overall network overhead. In addition to compressing the MA state using compression algorithms, it is worthwhile investigating alternative techniques for further reducing the volume of transferred data. As a result, a number of experiments/measurements have been conducted and several possible optimisations have been identified; when performing these optimisations, significant cost savings in terms of the state size may be achieved. A list of potential optimisations follows:

(a) Reduce the number of non-transient objects: Only the objects whose values are subject to change at each visited host, i.e. the objects that represent the knowledge obtained by an MA during its lifecycle, should be declared as normal (non-transient). The remainder variables should be declared as *transient* and assigned an initial value. Each time the MA is re-instantiated (de-serialised), the transient variable is assigned the same value specified by the programmer, even if that value was changed during the MA's execution on a previously visited host. In other words, when applicable, an object value should be '*hard-coded*' within the MA class. That decreases the degree of flexibility and autonomy given to the MA, as its behaviour/decisions cannot depend on the value of the transient objects, but saves the potentially unnecessary transfer of information related to the value of these objects.

(b) Use small number of characters for variable names: As noticed from Table B.1, serialised MA state carries information regarding the non-transient *variable names* (in addition to their corresponding values). By shortening the length of these names (e.g. using abbreviations instead of long, fully descriptive names) the amount of bytes required to encode this information would be reduced.

(c) Use small number of characters for MA class names: The MA state also embodies information referring to the corresponding MA *class name* as well as the *package* this class belongs to. Similarly to the previous optimisation, by assigning brief names to both the MA class and their packages, a few extra bytes could be saved.

[1] Objects or variables declared as 'transient' are not taken into account during an MA's serialisation process. When the MA state is de-serialised, transient objects are always assigned their initial values, namely they do not have any 'memory' of their value at the time the MA was serialised.

(d) Use primitive instead of complex data types: As described in Section 3.5.2, during the serialisation process, referenced objects are processed recursively until all non-transient objects are serialised. When not using primitive data types, but complex data structures for non-transient objects, improved flexibility is offered at the expense of increased state size, as these structures typically contain references to other objects. It is therefore preferable to choose primitive types whenever this is feasible, for instance to use *arrays* instead of `java.util.Vector` structures to store a list of values, to represent integer variables with *int* rather than with powerful and complex `java.lang.Integer` objects, etc.

Modification	**Size increment for non-compressed state (bytes)**	**Size increment for compressed state (bytes)**
Use a java.util.Vector structure instead of String[]	32	5
Use java.lang.Integer object instead of int	19	10
Use 20 characters string instead of 3 characters string	17	15
Include an additional 10 characters-long non-transient *String* variable	47	23
Include an additional non-transient *int* variable	17	10
Include an additional non-transient java.util.Vector variable containing a 10 characters-long string	79	28
Include an additional non-transient String[] variable containing a single 10 characters-long string	49	29

Table B.1. The effect of applying various source code modifications on the MA's state size

To illustrate the effect of the optimisations discussed above, we have measured the difference experienced in state size when applying the following modifications. These measurements are presented in Table B.1.

```
¼φ ¦  sr %MobileAgentPackage.MobileAgentExampleF+¦ σ? §? O  ¦ Z ¦ doTaskZ > encryptDataI
? hopL dataFoldert ?Ljava/util/Vector;L itineraryt ?Ljava/util/Vector;L ¤  originatingHostt
?Ljava/lang/String;xr (MobileAgentPackage.MobileAgentSuperclass ?  =+ $&·°]?   ? Z ? tcpL creatort
?Ljava/lang/String;L ?  idt ?Ljava/lang/String;L ?  itt ?Ljava/util/Vector;L ? seqt
?Ljava/lang/String;xpt ?  plato.essex.ac.ukt plato:0sr ?  java.util.Vector+ ù}[Ç;»?  ?   ? I
?  capacityIncrementI ? elementCount[ ? elementDatat ? [Ljava/lang/Object;xp?  ur
? [Ljava.lang.Object;?+ Xƒ?  s)l?   xpq ~pppppppppt ? sequenceNumber?   sq ~ ?  uq ~ ¤
pppppppppq ~ ? q ~
```

Figure B.1. The content of a MobileAgentPackage.MobileAgentExample class instance (object's state) as shown when printed on the standard output.

Figure B.1, illustrates the content of the *byte* array including the state of the MA presented in Figure B.2a, as it appears when printed on the standard output. Although several characters seem meaningless, it is clear that long package, class and non-transient variable names occupy more space in the serialised representation of the MA object state, than short names.

Significant reductions in the overall state size may be achieved by applying some of the optimisations outlined above. Given that the state size is typically not more than a few hundreds of bytes, it is clear that by summing up the savings obtained by the individual optimisations may lead to trimming a significant portion of the state size.

The proposed optimisations are illustrated in Figure B.2. The source code of an MA object with ‘non-optimised’ state is listed in Figure B.2a, with its ‘optimised’ counterpart presented in Figure B.2b. It can be noticed that in the latter case, several variables have been declared as transient, simpler data structures are used, whilst variable, class and package names have been shortened. Through applying all these optimisations, the size of the MA state is reduced from 628 bytes (391 when compressed) down to 333 bytes (227 when compressed). That corresponds to a reduction of 47% (41.9% when compressed).

```
package MobileAgentPackage;

public class MobileAgentExample extends
                                MobileAgentSuperclass  {
  Vector itinerary;
  Vector dataFolder = new Vector();
  String originatingHost;
  boolean encryptData;
  boolean doTask = true;
  int hop = 0;

  public MobileAgentExample (Vector it, String host,
                                  boolean encrypt) {
    itinerary = it;
    originatingHost = host;
    boolean encryptData = encrypt;
  }

  public void onArriving () {
    hop++;
    doTask = (hop % 2 == 0) ? true: false;
  }
  public void run () {
    System.out.println ("Hop number: " + hop);
    if (doTask)
      Task();
  }

  void Task() {
  }
}
```

(a)

```
package MAPack;

public class MAExample extends MA  {
  String[] it;
  String[] data;
  transient String origin = "plato.essex.ac.uk";
  transient boolean encryptData = true;
  transient boolean doTask = true;
  transient int hop = 0;

  public MAExample (String[] itinerary) {
    it = itinerary;
  }

  public void onArriving () {
    hop++;
  }

  public void run () {
    System.out.println ("Hop number: " + hop);
    if (doTask)
      Task();
  }

  void Task() {
  }
}
```

(b)

Figure B.2. Example source code of an MA with (a) ‘non-optimised’ vs. (b) ‘optimised’ state size.

It is clear though that transient objects do not allow the MA to make autonomous decisions based on their value. For instance, the MA corresponding to Figure B.2a will execute the `Task()` method only every second hop, while that shown in Figure B.2b cannot make such a decision based on the value of ‘*hop*’ variable, as that is assigned the same value *hop = 0*, every time the MA arrives at a new host. In addition, non-transient objects make it easier for user applications to interact with the MA and dynamically change its state content at runtime. For instance, in Figure B.2a, the user passes to the MA, through its constructor, the name of the originating host to whom the MA will return when completing its itinerary; the name can vary, depending on the host that creates the agent. In contrast, in Figure B.2b the name of the originated host is hard-coded within the MA bytecode and not transferred within the MA state. That implies an inflexible design, which is preferable though in the case that MAs are always created by a single device. However, should that device changes, the originating host name will need to be changed accordingly, that is the MA class will have to be modified and then *re-compiled*. Concluding, the choice of whether to declare an object as transient or non-transient represents a trade-off between flexibility and migration cost.

In the current framework implementation, *none* of the optimisations described above is applied. For instance, as far as variables and MA class names are concerned, relatively long and, therefore, more meaningful and descriptive names have been chosen, thereby introducing an additional migration penalty. Also, no effort has been put to reduce the number of non-transient variables so as not to affect the flexibility of dynamically modifying the MA functionality. That decision was dictated by the need of conducting a variety of experiments/ measurements under varying conditions, created by modifying several MA-based application parameters. It was therefore important to represent these parameters as non-transient objects in order to allow user applications to easily and instantly modify these parameters, e.g. through a Graphical User Interface (GUI), without going through a shutting_down_the_application/source_modification/compilation/starting_the_application procedure. Yet, as soon as such experimental investigations are finalised, optimal parameter values should be proposed and subsequently hard-coded so that the resulting MA state size is minimised.

B.3. MINIMISING THE LATENCY OF MOBILE AGENT TRANSFERS

In addition to reducing MA state size, minimising the delay associated with MA transfers is also of major importance, especially when considering time-critical management tasks. The investigation of the migration process itself is presented in Appendix C, where the MA transfer is broken down to several successive phases and ways of accelerating them are identified.

As mentioned in Section 4.4, MA transfers may be realised either over TCP or UDP as transport protocols. Hence, this section aims at investigating ways for optimising these protocols performance. The protocol parameters that may be customised through Java network programming are limited to the following:

(a) Stream buffering: The hierarchical nature of the I/O class library (`java.io` package) allows the programmer to build up streams in hierarchical manner. This is exemplified in the following code fragment taken from the TcpListener class of the MAL component (see Figure 4-9):

```
InputStream is = connection.getInputStream();
MasObjectInputStream dataIn = new MasObjectInputStream
                              (new GZIPInputStream(new BufferedInputStream(is)));
```

Since the getInputStream() method of the java.io.Socket class returns an java.io.InputStream object, the latter has been wrapped by a java.io.BufferedInputStream instance, which is in turn wrapped by a java.uil.zip.GZIPInputStream object. The main advantage of the BufferedInputStream class is that it enables data buffering. Without buffering, data would be read byte-by-byte, thus degrading performance [TAC00]. The buffer size may be specified, otherwise the default size is used. The default size is large enough for most purposes. A similar approach is followed for MAL's UdpListener class, where the MA state is retrieved and read from a byte array.

(b) TCP "No Delay" option: This optimisation refers to MA transfers through TCP. Since the size of transferred data is relatively small, the TCP *NODELAY* option is set on the client side. Without setting the TCP NODELAY option, the communicating parties activate Nagle algorithm [TAN96], which buffers incoming requests until the preceding request is acknowledged. On high-speed networks the use of Nagle algorithm can unnecessarily increase latency. TCP NODELAY option is set by both the sender and the receiver site following the establishment of a TCP connection, using the instruction:

```
connection.setTcpNoDelay(true);
```

(c) Further optimisations: In addition to configuring transport protocol parameters, as suggested above, there are also several improvements that may be performed to speed up the MA migration process. For instance, minimising the MA state size through adopting the optimisations suggested in Section B.2 would positively affect the migration latency, as it would considerably reduce the volume of transferred data. A second possible optimisation would be to include IP addresses rather than host names in the MAs itinerary vector. That would save the time needed to access the Domain Name Server (DNS), in other words, it would obviate the need to translate the names to their corresponding IP addresses prior to MA transfers.

Optimisations (a) and (b) have been already applied in our framework implementation while (c) is included in the list of potential future extensions.

APPENDIX C

MOBILE AGENT MIGRATIONS TIMING EXPERIMENTS

C.1. INTRODUCTION

In this appendix, we present the result of timing experiments that aim at providing a better understanding of the response time measured for Mobile Agent (MA) migrations. The objective is to investigate how the overall time is distributed among the individual phases of an MA migration and what is the effect of a number of factors such as the transport protocol used and the MA state size. In particular, the migration process is first divided into several independent, successive phases in order to assess the weight of their respective delays on the overall latency.

C.2. MOBILE AGENT MIGRATION TIMING MEASUREMENTS

The MA used for this experiment, termed 'ping-pong' MA, is launched by the manager application, visits a single remote Mobile Agent Server (MAS) server and then returns back to the manager, *without* performing any kind of task. Therefore, the phases identified are:

1. MA object creation (instantiation);
2. MA object serialisation by the manager application;
3. MA transfer to the MAS server;
4. MA object de-serialisation at the MAS side;
5. MA object serialisation by the MAS server;
6. MA transfer back to the manager station;
7. MA object de-serialisation at the manager side.

Both TCP and UDP transport protocols are considered for the MA transfers, while the serialised MA state can either be compressed before its transmission or non-compressed. Another investigation issue is the effect of the serialised MA's state size on the overall migration delay. In particular, one of the experiment's objectives is to examine to what extend does the increment of the state size affects the distribution of the individual migration phases delay. The average times corresponding to each migration phase are presented in Table C.1 and Table C.2, with the former corresponding to an MA object with relatively moderate state size (476 bytes when compressed, 678 bytes when uncompressed) and the latter to an MA with larger state size (1152 bytes when compressed, 3970 bytes when uncompressed). The state size growth is realised by increasing the number of the MA's non-transient variables [ARN96].

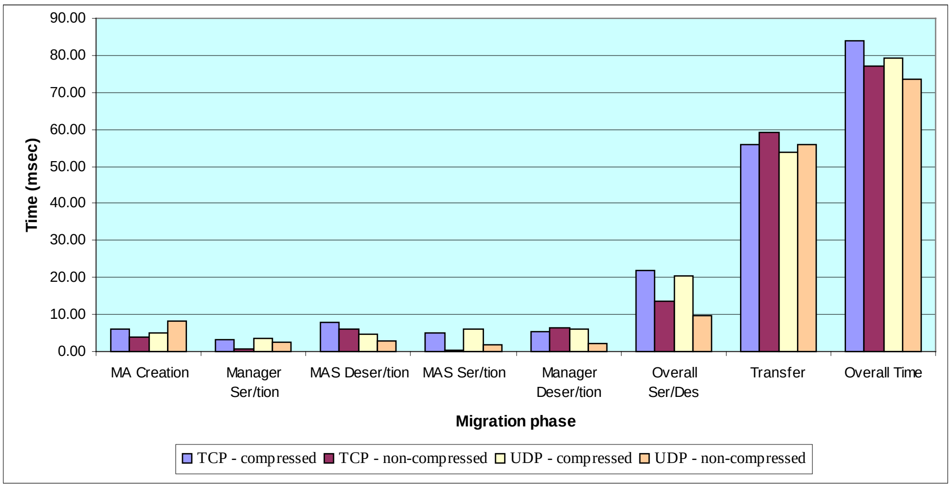
Migration Phase	*MA Creation*	*Manager Ser/tion*	*MAS Deser/tion*	*MAS Ser/tion*	*Manager Deser/tion*	*MA Transfer*	*Overall Time*
TCP, compressed	6.12	3.24	8.06	5.19	5.37	56.04	84.02
TCP, non-compressed	4.08	0.74	6.22	0.46	6.32	59.15	76.98
UDP, compressed	4.92	3.67	4.56	6.02	6.23	53.71	79.10
UDP, non-compressed	8.28	2.66	2.89	1.74	2.31	55.79	73.68

Table C.1: Time measurements (in msec) depicting the distribution of delays for a "ping-pong" MA with a 'small' state size (476 bytes compressed / 678 bytes uncompressed) during its lifetime (each table value represents 100 individual time measurements)

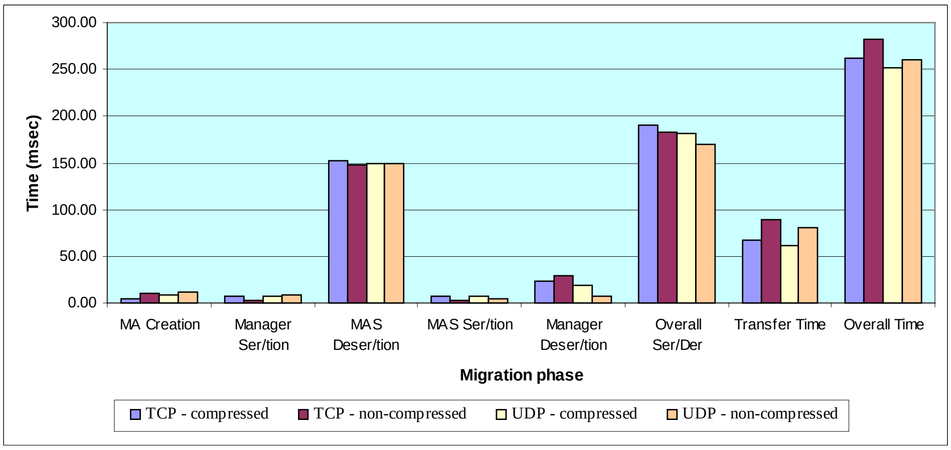
Migration Phase	*MA Creation*	*Manager Ser/tion*	*MAS Deser/tion*	*MAS Ser/tion*	*Manager Deser/tion*	*MA Transfer*	*Overall Time*
TCP, compressed	4.82	8.03	151.51	7.41	23.97	66.68	262.43
TCP, non-compressed	9.81	3.09	148.13	2.84	29.10	89.20	282.17
UDP, compressed	8.34	7.12	149.42	6.60	18.43	61.31	251.22
UDP, non-compressed	11.22	8.85	149.25	4.28	7.11	80.18	260.88

Table C.2: Time measurements (in msec) depicting the distribution of delays for a "ping-pong" MA with a 'large' state size (1152 bytes compressed / 3970 bytes uncompressed) during its lifetime (each table value represents 100 individual time measurements)

The main reason for conducting time measurement experiments with 'ping-pong' rather than with single-hop MAs is that it would not be feasible to measure MA transfers over UDP, as explained in Section 4.6.1.1. It is noted that the column representing the transfer delay represents the summation of the time needed to transfer the MA object from the manager host to the MAS site and vice-versa.



(a)

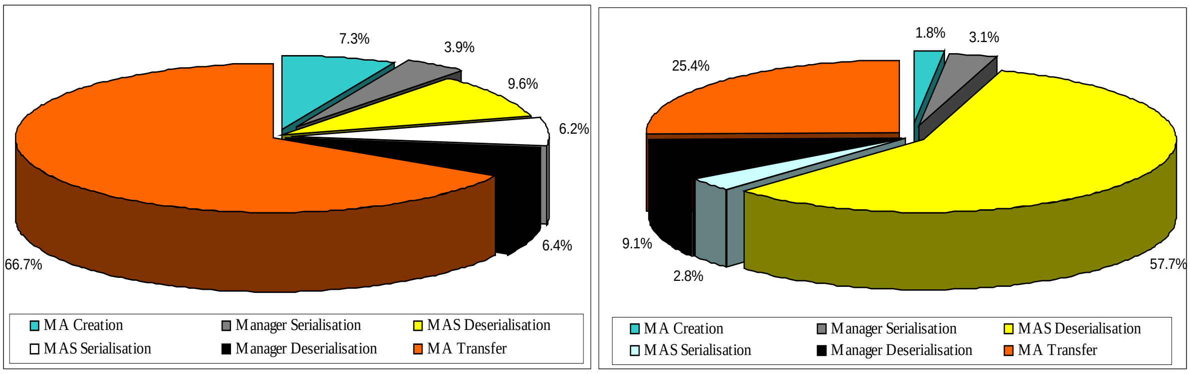


(b)

Figure C.1. Comparison of the delays experienced within the individual migration phases of an MA with (a) moderate or (b) large state size, depending on the transport protocol and on whether the serialised state is compressed prior to its transfer

The distribution of the overall response time among the phases that compose the migration process are graphically illustrated for both the types of MAs used in these experiments, i.e. with moderate (Figure C.1a) or large state size (Figure C.1b). It is interesting to observe the way that compression affects the response time as a function of the volume of compressed data. In particular, regardless of the utilised transport protocol (TCP or UDP), MA’s state compression always causes reduction of transfer time (the volume of transmitted data is decreased) and increment of the overall serialisation/se-serialisation time (compression/de-compression is regarded as part of the serialisation/se-serialisation process). However, the effect of compression on the overall response time largely depends on the MA’s state size. For serialised state of moderate size, compression increases the overall time. Conversely, when the migration of MAs with larger state sizes is considered, compression results in a reduction of the response time. Therefore, compression reduces the latency of the migration process in case that the original size of the serialised state and the compression ratio is such so as to justify the time penalty involved in the compression process. To illustrate, in our first type of MA, compression saves only 678-476 = 202 bytes from being transferred through the network (29.8% of the original data volume), while in the second type the saving becomes 3970 - 1152 = 2818 (71% of the original data volume).

Another interesting aspect of the MA migration time measurements is the alterations that state size causes on the distribution of the overall response time among the individual migration phases. The 'pie' diagrams shown in Figure C.2 confirm that increasing the MA's state size results in shifting response time "centre of mass" from transfer to the serialisation/de-serialisation process. For instance, the transfer latency of an MA with moderate state size covers the 66.7% of the overall response time, with the sum of the individual serialisation/de-serialisation times comprising the 26.1%. In contrast, when dealing with the MA with large state size, these percentages become 25.4% and 72.1% respectively.



(a) (b)

Figure C.2. Distribution (percentage of the overall response time) of the delays incurred within the individual migration phases for MAs with either (a) moderate or (b) fairly large state size (the serialised state is compressed)

Interestingly, de-serialisation process is proved more time consuming than its inverse process of serialisation. In addition, a significant portion of the response time is covered by the MA's state de-serialisation, which takes place at the MAS server side (see Figure C.2b). This is due to the fact that MAS entities use the customised ClassLoader (MACL) described in Section 4.4.3, which is slower than the default JVM ClassLoader used by the manager application. The serialisation process taking place at the MAS server is significantly faster than the de-serialisation, since it uses the MA class definition that has been already loaded by the MACL during the serialisation phase.

In general, the response time difference observed between the two types of MAs is mainly due to the difference in the overall serialisation/de-serialisation time, as shown in Figure C.3. Certainly, the transfer time also increases in the case of the MA with larger state size; however, this is not the decisive factor.

A possible optimisation would be to save the time needed to create (instantiate) the MA object, as suggested in Appendix B. That could be realised through creating the MA beforehand, that is, before the actual management operation commences. Such an optimisation would result in performance gain, especially in the case of MAs with small state size, when MA creation represents a considerable proportion of the overall time (7.3%).

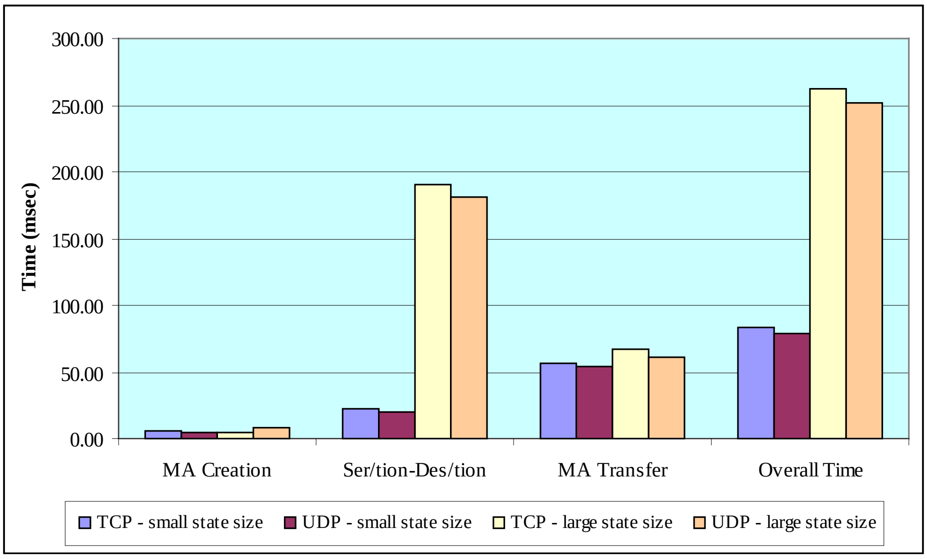


Figure C.3. Comparison of the individual migration phases delays for MAs with moderate vs. large state size (the serialised state is compressed prior to its transmission)