WEB-BASED MANAGEMENT OF IP NETWORKS AND SYSTEMS

THÈSE N^{o} 2256 (2000)

PRÉSENTÉE AU DÉPARTEMENT DE SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jean-Philippe MARTIN-FLATIN

Ingénieur généraliste diplômé de l'ECAM (Lyon, France)
de nationalité française

acceptée sur proposition du jury:

Prof. Jean-Pierre Hubaux, directeur de thèse
Prof. Roland Balter, corapporteur
Dr. Subrata Mazumdar, corapporteur
Prof. André Schiper, corapporteur
Prof. Morris Sloman, corapporteur

Lausanne, EPFL
2000

ABSTRACT

The management of IP networks and systems is currently based on the Simple Network Management Protocol (SNMP) and the SNMP management architecture. This poses a number of problems. Some are related to the efficiency, scalability, latency, and expressiveness of SNMP, others to the way the design of SNMP-based management platforms historically evolved. After reviewing the numerous alternatives that are currently investigated by the research community, including mobile code and intelligent agents, we propose to base the next generation of management applications on a new management architecture: WIMA, the Web-based Integrated Management Architecture. WIMA is based on standard Web technologies. It relies on a push-based organizational model for regular management (i.e., data collection—for offline processing—and monitoring over a long period of time) and notification/event delivery, and a pull-based organizational model for *ad hoc* management (data retrieval over a very short time period). Its communication model is characterized by (i) the use of persistent HTTP connections between agents and managers (or between mid- and top-level managers in distributed hierarchical management); (ii) the support for any information model (SNMP, CIM, etc.); and (iii) a reversed client-server architecture that facilitates crossing firewalls. In WIMA, the preferred method for representing management data in transit is XML. It is well suited for distributed hierarchical management; it unifies the communication model across the entire range of integrated management (that is, network, systems, application, service, and policy management); and it offers high-level semantics to the management-application designer. All the major problems that we identified in SNMP are solved in WIMA. Our architecture is validated by a prototype: JAMAP, the JAva MAnagement Platform.

Keywords: Network Management, Systems Management, Web-Based Management, Internet, Web, Push, WIMA, SNMP, CIM, HTTP, MIME, XML

Index Keys in the Computing Research Repository (CoRR):

- *Area*: Computer Science
- *Subject Classes*: Networking and Internet Architecture; Distributed Computing

RÉSUMÉ

La gestion des réseaux et des systèmes IP repose actuellement sur le protocole SNMP (*Simple Network Management Protocol*) et l'architecture de gestion SNMP. Ceci pose un certain nombre de problèmes. Certains sont liés à l'efficacité, au passage à grande échelle, à la latence et à la puissance d'expression sémantique de SNMP; d'autres sont dûs à la façon dont les plateformes de gestion SNMP ont vu leur conception évoluer avec le temps. Après avoir passé en revue les nombreuses alternatives à SNMP actuellement envisagées par la communauté de recherche, et notamment le code mobile et les agents intelligents, nous proposons de bâtir la prochaine génération de solutions de gestion sur une nouvelle architecture de gestion: WIMA (*Web-based Integrated Management Architecture*). WIMA est fondée sur les technologies Web. Elle repose sur un modèle organisationnel diptyque utilisant d'une part un modèle *push* pour la gestion régulière (c'est-à-dire la collecte de données à des fins d'analyse *a posteriori* et le *monitoring*, tous deux opérant sur une longue durée) et l'envoi de notifications/évènements, et d'autre part d'un modèle *pull* pour la gestion *ad hoc* (gestion au coup par coup, sur une durée très courte). Le modèle de communication de WIMA est caractérisé par (i) des connexions HTTP permanentes entre agents et gestionnaires (ou entre gestionnaires de niveaux supérieur et inférieur en cas de gestion distribuée hiérarchique); (ii) le support de n'importe quel modèle informationnel (SNMP, CIM, ou autre); et (iii) une architecture client-serveur inversée qui facilite la traversée des pare-feux. Dans WIMA, il est recommandé d'utiliser XML pour représenter les données de gestion en transit. XML est bien adapté à la gestion distribuée hiérarchique et permet d'unifier le modèle de communication à travers le spectre entier couvert par la gestion intégrée: gestion de réseaux, de systèmes, d'applications, de services, de politiques, etc. XML offre également un niveau sémantique élevé au concepteur d'applications de gestion. Tous les problèmes majeurs que nous avons identifiés dans SNMP sont résolus dans l'architecture de gestion WIMA. Notre architecture est validée par un prototype: JAMAP (*JAva MAnagement Platform*).

Mots-clés: gestion de réseaux, gestion de systèmes, gestion basée sur les technologies Web, Internet, Web, *push*, WIMA, SNMP, CIM, HTTP, MIME, XML

Classification dans le *Computing Research Repository* (répertoire international de travaux de recherche en informatique):

- *catégorie*: informatique
- *sous-catégories*: réseaux et architecture de l'Internet; informatique répartie

ACKNOWLEDGMENTS

I would like to thank Jean-Pierre Hubaux for welcoming me on his team and supervising my Ph.D. thesis for four years. Having spent many years in industry, I dreaded becoming a student again. But he patiently put up with my frustration with no longer being in charge, he offered me exceptional working conditions, and he got the best out of me. I am very grateful for the freedom and independence he gave me over these years, and for helping me complete this dissertation on time. I also thank him for arranging the funding of my work by the Swiss National Science Foundation (FNRS) and EPFL.

I am grateful to all the members of my Ph.D. committee for the time they spent reviewing my dissertation and the valuable feedback they provided me with.

I also thank all my current and past colleagues at EPFL (especially those at ICA, TCOM, LIA, and LSE) for the friendly atmosphere and the good times we spent together. I especially thank Simon Znaty, who taught me how to do research and gave me a goldmine of pointers in network management, and Werner Almesberger, who helped me understand how the Linux kernel works and gave me some precious feedback on Chapter 7. Thanks also to Danielle Alvarez, our ever-smiling secretary, for shielding me from administrative headaches for four years; and thanks to Holly Cogliati for patiently improving my English and teaching me the differences between British and American English.

During my Ph.D., I had the chance to meet several people outside EPFL with whom I had regular technical discussions, face to face or via email. They all contributed to improving the quality of my work. Special thanks go to Luca Deri, George Pavlou, Gian Pietro Picco, and Jürgen Schönwälder. I also thank the people I met at AT&T Laboratories, Bell Laboratories, IBM T.J. Watson Research Center, Imperial College, Institut d'Informatique et Mathématiques Appliquées de Grenoble (IMAG), and University of Twente for the multitude of questions they asked me while I was giving seminars there; they helped me consolidate my work.

My Ph.D. work would not have been possible without the technical skills that I developed in my previous job. I would like to thank my former colleagues at ECMWF: Ditlef Martens, Otto Pesonen, and Baudouin Raoult, who taught me so much about Unix internals; Tony Bakker and Dick Dixon, who taught me the basics of networking; Walter Zwieflhofer and Claus Hilberg, my former supervisors, who taught me rigor when selling a technical solution to management; and all the others (Pam Prior, Petra Kogel, Matthias Nethe, Mike O'Brien, Hélène and Didier Garçon, etc.) who made my six years in England such a great working and human experience in my life.

Further back in time, I am very grateful to Patrick Lachaize, who encouraged me to move into networking in 1989, before I heard of the Internet. What a visionary idea! I also express my deepest gratitude to Gilles Maignaud, Alain Martinez, and François Martin for teaching me, when I was 16–18, that mathematics, physics, and science at large can be great fun. Even in difficult times, I have always strived to take some pleasure in

doing my job. Their advice has served me throughout my career, and I am indebted to them for my continued interest in computer science.

Finally, I thank my family and friends for their unwavering support throughout these four Ph.D. years. When other people were telling me that I was crazy to go back to university, they supported my project and encouraged me to pursue my childhood’s dream of becoming a researcher. Most of all, I thank my wife, Béatrice, whose daily love made my Ph.D. possible. She supported me when I had difficulties and offered me the family cocoon that enabled me to be creative. She also put up with the long nights and week-ends that I spent at the university and allowed me to commit fully to my work. I knew that I had married a rare pearl...

Lausanne, August 11, 2000

To Béatrice

To the memory of Louise and Henri

Γηράσκω δ'ἀιεὶ διδασκόμενος

The older I get, the more I learn

Solon (~640 – ~558 B.C.)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

A peculiar property of network and systems management is that it looks pretty simple at first sight, at least in the IP[1] world. But as soon as you set about implementing or using a management application, you unveil problem after problem and gradually unfold the *Big Book of Management Hassles*. After spending several years on this topic, most reasonable people decide to turn to other problems perhaps more technically difficult, but also less multifaceted, and in the end less complex and less challenging. The author decided to do otherwise and endeavored to spend four years of his life to find a better solution to manage IP networks and systems. Whether any conclusion can be drawn about his sanity is left for the reader to decide... Anyhow, what will now unfold before your eyes are the technical details of his proposed solution, a new management architecture for the next decade, and an attempt to convince you of its relevance to the IP world.

1.1 Background

Throughout the 1990s, the management of IP networks has relied almost exclusively on a single protocol, the Simple Network Management Protocol (SNMP), and a single management architecture, confusingly also called SNMP. The primary achievements of SNMP were its simplicity and its interoperability. SNMP-based agents were easy to develop and integrate into managed devices. As a result, proprietary network management solutions gradually gave way to open solutions. Whatever the equipment vendor, whatever the type of equipment, you could manage it with a single third-party management platform such as HP OpenView, Cabletron Spectrum, IBM/Tivoli Netview, Sun Microsystems Solstice, etc.

The market of IP networks management first went through a period of great and fast development; during 1990–95, SNMP gradually became ubiquitous. In 1990, the RFCs specifying what is now called SNMPv1 were issued by the Internet Engineering Task Force (IETF). In 1993, only three years later, it was commercially mandatory for a vendor to support SNMP on its entire range of network equipment, from the top-of-the-range backbone IP router to the bottom-of-the-range print server, because many customers made the support for SNMP a requirement in their requests for bids.

1. Many acronyms used in this introductory chapter will be expanded and explained in the next two chapters, when we investigate SNMP and its potential alternatives.

The situation changed in the mid–1990s. SNMPv2 raised high hopes, but proved to be a technical and marketing failure [141, 166, 207, p. 334]. By the time SNMPv3 was issued, in 1998, the market no longer believed in the capacity of SNMP to evolve and meet its needs. The relevance of traditional SNMP-based management was questioned by new vendors who wanted to enter this lucrative market, and were looking for ways to undermine the near monopoly of the four main management-platform vendors (HP, Cabletron, IBM, and Sun Microsystems). They began to offer new platforms based on new technologies that were appealing to the market (e.g., AdventNet with Web technologies). SNMP-based management was also questioned by customers, who were scared by the soaring costs associated with it. Customers also wanted to integrate network management with systems management. But while IP networks were mostly managed with open SNMP-based management platforms, IP systems, conversely, were often managed with proprietary non-SNMP platforms (e.g., Novell NetWare or IBM/Tivoli TME for Windows-based PCs, or RPC-based management platforms for Unix workstations). Equipment vendors also questioned SNMP-based management because their device-specific management GUIs (Graphical User Interfaces) had to be ported to an ever-growing number of management platforms and operating systems, which made their development costs sky-rocket. Most of all, the relevance of SNMP was undermined because the problem at stake was becoming more and more different from the problem solved in the late 1980s. Today, virtually all devices are networked; most of them embed a reasonable amount of processing power; some devices are wireless; data and telephone networks are merging; customers demand security for both Local-Area Networks (LANs) and Wide-Area Networks (WANs); customers want to integrate network, systems, application, service, and policy management; the TCP/IP stack is everywhere, in every system and every network device; Web technologies are used everywhere; etc. None of these assertions were true when SNMPv1 was devised. No wonder the solution selected in the 1980s for slow and small networks, with mostly low-profile agents, no longer appeared adequate to the market!

In the second half of the 1990s, the main change in Network and Systems Management (NSM) was that many newcomers looked at it with software-engineering eyes, with no background (hence no habits) in NSM. Some of them originated from the object-oriented world, others from the artificial-intelligence world, yet others from the database world, to name a few. And when these people analyzed how IP networks are typically managed today (that is, how management platforms are designed, how efficient SNMP is as an access protocol, whether the principle of data polling inherent to the SNMP management architecture is efficient, etc.), they soon realized that most SNMP-based management applications do not withstand the comparison with modern distributed applications. Why not use object-oriented analysis, design, and implementation, which are widely adopted by the software industry today? Why be limited by the few existing SNMP protocol primitives to collect data from an agent? Why incur the network overhead of having the manager repeatedly tell every agent what set of Management Information Base (MIB) variables it is interested in, when this selection remains fairly constant over time? Why not compress management data when it is transferred in bulk between agents and managers? Why use an unreliable transport protocol to send a critical notification to a management station when an interface goes down on a backbone router? Why make it so difficult to manage remote subsidiaries across firewalls? Why concentrate all the management application processing at the manager and let the agents do nothing?

The software-engineering community had many alternatives to offer: Web technologies, mobile agents, active networks, CORBA, intelligent agents, and so on. This resulted in a plethora of new proposals that often departed completely from the design decisions made a decade earlier. This also utterly confused the market. Customers like simple messages, like “Buy SNMP because everyone does” or “Choose between CORBA and DCOM”. They make their investments strategically safe and easy to justify to top management. But the market was telling them: “You can use mobile agents in Java, or Tcl/Tk, or Telescript... They are not mature yet, but they will be soon and they are technically very attractive. You can also have CORBA, and DCOM, and Enterprise JavaBeans (EJBs), and intelligent agents speaking KQML or ACL, and cooperative object-oriented distributed platforms communicating via CORBA or Java RMI or...”. As expected, customers were (and still are) very reluctant to go from a very homogeneous and strategically safe market to such a hectic and hazardous one. Some people decided to go for the most powerful: they became Microsoft-only shops and adopted closed solutions based entirely on Windows and DCOM. Others preferred to put their eggs in several baskets; they kept several brands of material, they stuck to open systems, and put up with the inconvenience of supporting

multiple management systems: one for network devices, one for Windows PCs, one for Unix systems, one for the intranet RDBMS, and so forth. By and large, most customers have decided to postpone heavy investments to better days, when the future of network, systems, application, service, and policy management would be clearer to read, and when *integrated management* would mean that all types of management could be integrated within a single platform. Today, investing in a management solution is a risky business.

It is our belief that NSM goes through cycles, alternating between periods when the market is clear to read, standards are stable, and most customers make similar decisions to solve a given problem, and periods when the market is obscure to read, many new standards are in the making, and it is obvious to neither vendors nor customers what will succeed in the next management cycle. The 1980s were the days of proprietary management solutions. The 1990s proved to be the days of SNMP for network management, and proprietary solutions for systems management. What will be the basis for the NSM solutions of the next decade?

1.2 Ph.D. Work

This Ph.D. work was initiated in mid–1996, as the market was becoming confused. Coming from industry, having managed networks for six years, and having thoroughly investigated the SNMP-based management-platform market at the end of 1994, the author was familiar with the strengths and weaknesses of SNMP-based management, and wanted to make his small contribution to the next management cycle. The purpose of this work was to study the management architectures and technologies that could be used for the next cycle, to propose a new technical solution, to demonstrate its feasibility and simplicity by developing a prototype, and to give a vision of how this solution could evolve during the next decade—that is, propose a migration path from current management solutions to more visionary solutions.

To achieve this goal, the approach adopted for this thesis was to study NSM with both a network-management hat and a software-engineering hat. Due to market pressure, most project managers in industry are forced to jump on the bandwagon of the latest technology and develop as quickly as possible a fast-designed solution, only to be the first on the market to support that technology. The rule of the market these days does not seem to be “Do it right”, but rather “Be there first and trumpet loud”. Being outside industry and immune to particular commercial interests, we took a different approach. We took the time to investigate most (hopefully all) management paradigms proposed to date, we critically analyzed them, and thought about potential improvements. We compared different software-engineering approaches in the specific context of NSM, selected one (Web-based management), improved it significantly, and proposed what we think is a good technical candidate for the management cycle of the next decade.

To a software engineer, IP networks appear to be fairly easy to manage because they impose no stringent requirements on the management application. Typically (that is, if we ignore the extreme cases such as embedded systems in spatial or military equipment), we have no real-time constraints, fault-tolerant systems are not required, and we can even afford to lose some management data (not too much). Therefore, from a software-engineering perspective, a management application is a fairly simple case of distributed application: we have one or several managers, and many agents (managed devices); most data goes from agents to managers; and we can process management data (e.g., compute usage statistics or correlate alarms) either in an agent, in a mid-level manager, or in a top-level manager. Similar tasks are routinely performed in other application domains, typically by relying on a Distributed Processing Environment (DPE, also called middleware) such as CORBA, EJBs, or DCOM, and by buying Commercial Off-The-Shelf (COTS) component software.

The complexity of NSM only appears at a later stage of analysis. In the IP world, it mainly stems from its tight constraints as far as scalability and resource usage are concerned. First, the number of nodes to manage can grow large and span multiple management domains—consider, for instance, a large, geographically dispersed enterprise. Second, the amount of management-related configuration data in a single device can also be important—e.g., the access-control lists in a router-based firewall. Third, we want to manage a wide range of heterogeneous equipment. A $150,000 backbone router is vastly different from a $50 print server. The relative

cost of management software is very different, compared with the overall cost of hardware and software; so are the CPU and memory resources available. Fourth, we want to do many things (reactive management, preventive management, security, accounting, billing, etc.) which, taken individually, are in general reasonably simple to implement and manage, but become complex when put together. Fifth, *all* resources allocated to management (memory, CPU, network bandwidth, manpower) are considered overhead, and should therefore be kept to a minimum. Sixth, changing the habits of a market takes time; when SNMP appeared on the market, it took two to three years to deploy a new generation of devices, with new hardware and software. Seventh, changing the habits of a market costs a fortune, so you do not want to make a mistake when you decide to invest in a new management solution. As an equipment vendor, if you embed the wrong management software in all your devices (that is, the market decides to adopt a different standard after tens of thousands of your devices have been sold), the cost is enormous for you to migrate, but also—and above all—for your customers, as they will have to install new software in equipment already deployed in production networks—the last thing you want to ask your customers to do. In other words, large vendors that have a reputation to save will not embark on a new technology before they are reasonably sure that the market is heading in that direction. As the number of candidate technologies is large, we face a chicken-and-egg situation. Once you combine all these problems, then it becomes obvious that NSM is more complex and challenging than it first appears.

As we mentioned before, there are many candidates to succeed SNMP in integrated management. In this dissertation, we explain why, in our view, Web-based management is the best candidate for the next management cycle. Our argument is sixfold. First, Web technologies allow us to solve most of the problems that we identified in SNMP-based management. Second, the solutions we describe in our dissertation are simple. They could be engineered and widely deployed in less than a year. Mobile agents, conversely, require environments that are both secure and fast (especially for WAN links), which no one can presently provide. Similarly, simple, yet efficient, multi-agent systems still remain to be seen in NSM. Third, Web technologies can have a very limited footprint on resource-constrained network devices, unlike CORBA. Fourth, Web technologies offer a smooth migration path toward the future currently envisioned by the communication-service industry, that is, dynamic service provisioning with mobile code. Fifth, Web-based management does not revolutionize NSM. It can cope with legacy SNMP-based systems, and can also smoothly integrate new information models such as CIM. Sixth and last, the Web has encountered a tremendous success in the enterprise world. Its simplicity, combined with the portability of Java, have made it ubiquitous, and it is difficult today to find a software-engineering field that is not leveraging it. Web expertise is rapidly developing worldwide, and it makes sense to capitalize on this wealth in NSM.

The idea of using Web technologies in NSM is not ours. It is not even new: experiments began in 1993–94 with Web browsers, HTTP, HTML, and CGI scripts. The novelty is our management architecture called WIMA (Web-based Integrated Management Architecture). Its main contributions are the following:

- an organizational model based on push technologies, which unifies the transfer of notifications and regular management data from agents to managers (or between managers in distributed management);
- a communication model that is totally independent of the information model; and
- the use of XML to distribute the management application across a hierarchy of managers, throughout the range of integrated-management tasks.

1.3 Caveat

During the numerous seminars that the author had the opportunity to give during his Ph.D. work, there was a standard misunderstanding about the type of networks and systems that this work addresses. In order to avoid a similar misinterpretation of this dissertation, the reader is invited to pay attention to the following *caveat*:

> This work deals with the management of IP networks and systems typically used today in computer networks, as opposed to OSI-based telecommunication networks typically used in classic telephony. Our conclusions apply equally well to standard IP data networks, IP multimedia networks, and IP telephony networks. Due to time constraints, we did not fully investigate wireless networks and therefore do not consider them here; but early work suggests that most of our conclusions apply to fixed (wireline) as well as wireless networks, especially our push-based organizational model. When we refer to network and systems management (or NSM for short) in this dissertation, we mean the management of fixed IP networks interconnecting IP network devices and IP systems.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we define our terminology and state the problem that we are striving to solve. In Chapter 3, we present an overview of the solution space in the form of two taxonomies of NSM paradigms. In Chapter 4, we analyze this solution space and select Web-based management with weakly distributed hierarchical management. In Chapter 5, we summarize the state of the art in Web-based management. In Chapter 6, we describe WIMA, our management architecture, and especially its push-based organizational model for regular management data. In Chapter 7, we detail WIMA-CM, the communication model of WIMA. In Chapter 8, we study XML and show that it can significantly contribute to the next management cycle, especially for distributed management and integrated management. In Chapter 9, we present JAMAP, a research prototype that was developed to demonstrate the feasibility and simplicity of the core ideas behind WIMA. In Chapter 10, we compare our solutions with others. Finally, we conclude and give directions for future work in Chapter 11.

Chapter 2

PROBLEM STATEMENT

In this chapter, we substantiate our claim that SNMP-based management is not appropriate for the next management cycle and justify why a new management architecture is needed. To do so, we recall how IP networks and systems are typically managed today and explain why this should change[1].

This chapter is organized as follows. In Section 2.1, we define the terminology used throughout this dissertation. In Section 2.2, we characterize implicit and explicit design decisions that were made in the SNMP management architecture. In Section 2.3, we review the main strengths of SNMP. In Section 2.4, we describe the main problems that we identified in SNMP-based management. In Section 2.5, we show that the problem solved by SNMP has changed; what we need today is a new solution to a new problem. Finally, we summarize this chapter in Section 2.6.

2.1 Terminology

Before we describe the problem at stake, let us first define our terminology. The need for this is due to the inconsistency in the terminology used by the NSM research community. What is a manager: a person or a program? What is an NMS: a program, a machine, or a group of machines? The situation is even worse when we build interdisciplinary teams, as mentioned in the introduction, and bring together people with different backgrounds. What is an *agent* for people coming from the Internet, telecommunications, software engineering, or Distributed Artificial Intelligence (DAI) communities? To address this confusion in this dissertation, we will adhere to the definitions given in this section.

We assume here that the reader is reasonably familiar with network and systems management, SNMP, the way SNMP-based management platforms work in the IP world, how they are typically structured, and the management tasks they perform (for an introduction, see Stallings [207] and Rose [176]). We therefore do not redefine here all the well-established concepts, but only those whose definition is not consensual and those that are specific to this work.

1. The author would like to thank Jürgen Schönwälder for his detailed feedback on this chapter.

2.1.1 IP world vs. telecom world

In Section 1.3, we implicitly defined two separate worlds for networking:

- The *IP world*, where all pieces of equipment support the TCP/IP stack[1] for communication. Some authors call it the *Internet world*. A typical example is a data network.
- The *telecom world*, where all pieces of equipment support part or all of the Open Systems Interconnection (OSI) stack for communication. A typical example is a traditional telephone network.

This thesis deals exclusively with the IP world, as suggested by its title.

2.1.2 Network, systems, application, service, policy, and integrated management

In the IP world, *network management* primarily deals with network devices, that is, equipment whose sole purpose is to make the network operate properly. Typical examples include IP routers, bridges, Asynchronous Transfer Mode (ATM) switches, level-3 switches, level-4 switches, intelligent hubs, and plain hubs. Some people also include in this category the network-accessed peripherals that are shared by a group of people (e.g., printers or disk servers), on the basis that they are accessed via the network. We do not, as a peripheral can usually be directly attached to a host and is useful *per se*: its usefulness does not depend on the network. In contrast, an IP router in itself is totally useless for an end user.

In the OSI world, there is a strong distinction between *element management*, which is the management of individual pieces of equipment, and *network management*, which is the management of an entire network abstracted as an entity of its own. For years, the IP world has been interested exclusively in the former. With the advent of concerns related to the Quality of Service (QoS) delivered by IP networks, the IP world has become increasingly interested in managing end-to-end network characteristics as well. As a result, network management in the IP world now encompasses both element management and network management in OSI parlance. In this dissertation, we will consider both of them when we refer to network management. The agent-manager interactions described in the next chapters will typically be about element management, whereas manager-manager interactions will generally be about network management (in the OSI sense).

In its simplest sense, *systems management* is concerned with hosts, that is, individual machines that can operate and fulfill useful tasks for users without a network. The objective of systems management is to make sure that the host runs smoothly. Typical areas of interest to systems management include the monitoring of the Central Processing Unit (CPU), memory, and disk usage; the management of processes, file systems, and access-control lists (see the Host Resources MIB [237]); and the detection of hardware faults. We include shared peripherals in this category.

A piece of equipment can either be classified as a network device or as a system: the former is dealt with by network management, the latter by systems management.

Application management is about managing programs that interact with users or other programs. These programs run on top of an operating system. The management of the programs belongs in the application-management realm, whereas the management of the underlying operating system belongs in the systems-management realm. Examples of applications that run on stand-alone machines and must be carefully managed are relational databases and object-oriented databases. Application management commonly deals with access control, performance monitoring, fine-grained buffer allocation, etc. The objective here is to ensure that the application performs correctly, and to detect and repair a problem before users complain. Systems management is more or less generic, whereas application management is inherently site specific.

---

1. The so-called *TCP/IP stack* does not only include the Internet Protocol (IP) and the Transmission Control Protocol (TCP), but also the User Datagram Protocol (UDP), the Internet Control Message Protocol (ICMP), the Simple Network Management Protocol (SNMP), etc.

Unfortunately, these nice, clear-cut definitions are not always valid. The situation becomes fuzzy when we consider distribution aspects. Initially, a *distributed system* was a group of hosts that collectively offered a service and appeared as a single machine to that service. An example of distributed system is a cluster, that is, a group of machines sharing multiaccess storage devices. This concept, popularized by Digital Equipment in the 1970s and 1980s, has been adopted since then by many Personal Computer (PC), workstation, and server vendors in the Windows and Unix worlds. A cluster allows for transparent load balancing and automatic recovery (*failover*); it provides applications with increased robustness and offers some kind of fault tolerance. Another example is a massively parallel system with hundreds of microprocessors. To the program running on it, the entire system appears to be a single machine[1].

Initially, a *distributed application* was an application running on several hosts, possibly on a distributed system. A distributed application consisted of several application modules, each of them running on a different machine on top of an operating system, itself running on top of the hardware. An example of such a distributed application is NIS (Network Information Service), a popular network lookup service developed by Sun Microsystems in the 1980s and now found on virtually all Unix platforms. NIS relies on different programs that run on different machines, but globally, it is used by the administrator as a single program.

Then came the main industrial outcome to date of the research efforts in distributed operating systems: *Distributed Processing Environments* (DPEs), also called *middleware*. A middleware sits between the operating system and the application. It can be viewed as a distributed application because it fulfills the above definition: it consists of several modules running on top of the operating system of different machines. But it can also be viewed as a distributed system, because it offers a pseudo operating system to the application running on top of it: this application need not be aware whether the operating system underneath is distributed or not. A well-known example is the Common Object Request Broker Architecture (CORBA). Another example is a distributed relational database: it is not an operating system, but it is not a real application that a user may interact with; it is a building block, on top of which a real application may run.

Things are even worse because frequently, in the literature, there is confusion between *distributed systems management* (that is, a distributed way of managing systems) and *distributed-systems management* (that is, the management of distributed systems). This confusion is often created purposely, when the authors want to encompass both.

Today, the expression *systems management* usually covers all of this: the management of stand-alone hosts, the management of tightly coupled groups of machines, the management of middleware, and the distributed way of managing systems. As a result, there is a large overlap zone between distributed systems and distributed applications—so much so that some people include application management in systems management. Kramer, for instance, uses these two expressions interchangeably [121]. For us, the distinction between the two is not an important issue, because systems and application management are very similar in terms of management architecture and communication protocol. Although application management is not explicitly covered by this thesis, most of our conclusions apply also to it.

Service management is not far from application management either, because most services are implemented by applications. The expression *service management* comes from the telecom world and only recently entered the IP world, whereas application management is more commonly used in computer science. However, there is an important nuance between the two. The term *service* implicitly refers to a contract between a provider and a customer. Thus, service management is mostly concerned with what the customers see: the Application Programming Interfaces (APIs) they interact with, the type of service they purchased (e.g., gold, silver, or economy), and the monitoring of the service actually delivered to ensure that the contract defined by the Service-Level Agreement (SLA) is fulfilled. Two important elements of service management are thus QoS management and SLA monitoring. Application management, conversely, is concerned with tuning the internals of the service (e.g., increasing the number of large memory buffers and decreasing the number of

1. Actually, there are some cases when we do not want to conceal all the details of the distributed system from the application. For instance, the programmer may want to know the number of processors available on the system to optimize the efficiency of the code.

small memory buffers in a memory pool used by a database). Application management deals with the nuts and bolts that make up services.

Service management is high level, application management is low level. Service providers keep application management internal, and hide it from customers and competitors. Customers do not see the nitty-gritty of the databases and infrastructure that underpin the provision of the service. But service providers do disclose data relevant to service management. Some of it is destined to their customers, to prove that the contracts have been honored (provider-side SLA monitoring). Some is sent to other service providers (e.g., when the provision of one service requires the availability of others, provided by other companies).

Policy management is concerned with the definition and enforcement of high-level management decisions. Another name for it is *policy-driven management* [196]. An example of policy is the following: in case of network congestion, the customers paying for a gold service get the absolute priority; if there is some bandwidth left, customers paying for a silver service get up to 90% of the remaining bandwidth, while customers paying for an economy service get the remainder. This policy has repercussions on service management (SLAs) and network management (configuration of queues in routers). More generally, policy management interacts with network, systems, application, and service management.

Finally, the objective of *integrated management* is to integrate network, systems, application, service, and policy management within a single, distributed management platform, as opposed to having one management platform per type of management.

In this dissertation, we are primarily concerned with the integration of network and systems management. But in some places, we will also make some observations relevant to application, service, and policy management. A large part of Chapter 8 will be dedicated to integrated management.

2.1.3 Management application, platform, and system

In network management, the management software is called the *Network Management Application*. In reality, several independent applications may be run to manage an entire network; in this case, we consider them collectively as a single distributed application. A Network Management Application is composed of *managers* running in *Network Management Stations*, and *agents* running in network devices (also known as managed devices). The terms *manager* and *agent* come from the manager-agent paradigm followed by the OSI management model and the SNMP management architecture—i.e., the piece of software is named after the management role played by the machine it runs on.

Likewise, in systems management, a *Systems Management Application* is composed of managers running in *Systems Management Stations*, and agents running in managed systems. These systems may be related (e.g., components of a distributed system) or independent (e.g., hosts in an intranet). The Network Management Application and the Systems Management Application are often, but not always, integrated. To keep the text fluid and remain generic, we will refer to the *management application* when we actually mean the Network Management Application, the Systems Management Application, or both. Similarly, a *management station* can be a Network Management Station, a Systems Management Station, or both.

A *management platform* is the manager side of a management application. It is characterized by a certain version of the manager (e.g., HP OpenView Network Node Manager 6.1), a certain version of the operating system of the management station (e.g., Linux 2.3.28), and the hardware of the management station (e.g., a Compaq Pentium III PC). A management platform may be specialized for network management (*Network Management Platform*), systems management (*Systems Management Platform*), or may integrate both.

The acronym *NMS* used to stand for *Network Management System*, in the early days of SNMPv1. But what exactly is a Network Management System? To some people, it is synonymous with Network Management Station; to others, it refers to the Network Management Platform; to yet others, a single Network Management System consists of all the Network Management Stations and the Network Management Application. To avoid any ambiguity, we will use neither the acronym *NMS* nor the expression *Network Management System*.

2.1.4 Agent vs. mobile agent vs. intelligent agent

By extension, the *managers* often refer to the management stations, and the *agents* refer to the managed devices or systems. These are clearly misnomers, for they confuse the management application running on a machine with the machine itself. But these terms are seldom ambiguous once placed in context.

To avoid any confusion between programs and people when we use the term *managers*, the people in charge of managing networks or systems will be called *administrators* (this convention comes from the IETF).

The meaning that we retained for the word *agent* is standard for the NSM community. It is inherited from the manager-agent paradigm, one of the building blocks of the OSI and SNMP management architectures. But we experienced that it is misleading to people coming from software engineering or DAI, because they routinely use it in a different sense. To avoid any confusion, an agent in the software-engineering sense will be called a *mobile agent*, which is essentially a technique to dispatch and execute code on a remote entity. Likewise, we will speak of an *intelligent agent* when we mean an agent in the DAI sense, that is, a paradigm enabling elaborate (and sometimes complex) forms of cooperation between remote entities that "think" independently of each other. We will come back to mobile agents and intelligent agents in Chapter 3.

2.1.5 Proxy vs. gateway

To cope with legacy systems whose internal agent does not support the capabilities expected by the manager, we assume hereafter that legacy systems make use of *management gateways* if necessary (see Fig. 1). A management gateway is generally dedicated to a certain legacy system, and external to it. It is located between the manager and the agent, and is transparent to the management application. It can, for instance, translate a CORBA request into SNMP protocol primitives, and *vice versa*. When a management gateway is used, the agent embedded in the legacy system is called a *legacy agent*. Throughout this dissertation, when we refer to an *agent*, we may refer either to the pair {legacy agent, management gateway} or to a single agent.

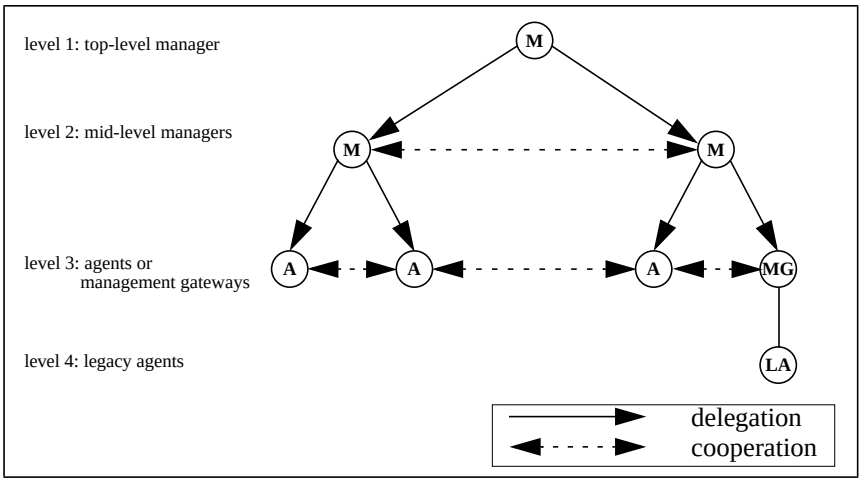

Fig. 1. Delegation and cooperation

The management gateway is called a *proxy agent* by some authors [97, 122]. The problem with this designation is that the concept of proxy is confusing and ill-defined. This was acknowledged by the IETF: "The term 'proxy' has historically been used very loosely, with multiple different meanings" [125, p. 4]. The definitions of a proxy proposed in Request For Comment (RFC) 2573 [125] and RFC 2616 [74] are very specific and different from our definition of a management gateway. Our choice of the term *gateway* was based on the definition given in RFC 2616 [74, p. 10]: "[A gateway is] a server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway". Therefore, we will not use the phrase *proxy agent*.

For the sake of completeness, we should also mention that a management gateway is sometimes called a *delegated agent* [254]. This phrase is ambiguous, as some authors give this name to programs remotely transferred to an agent [88]; so we will avoid it, too.

2.1.6 Delegation

Decentralized management is to the enterprise world what distributed management is to computer science: a management paradigm based on the delegation of tasks to other entities. These entities are people in the enterprise world, and machines or programs in computer science. *Delegation* is used in both contexts as a generic word to describe the process of transferring power, authority, accountability, and responsibility [71, 152] for a specific task to another entity. In distributed NSM, delegation always goes down the management hierarchy: a manager at level (N) delegates a task (i.e., a management processing unit) to a subordinate at level (N+1); this is known as *downward delegation*. In the enterprise world, we can also find *upward delegation*; for example, an employee delegates his tasks to his manager when he is out due to illness [152]. Downward delegation and upward delegation are two kinds of *vertical delegation*, typical of hierarchical paradigms. In the enterprise world, organization charts generally follow a hierarchical paradigm. They are characterized by a multi-layer pyramid, comprising a *top-level manager* (at level 1), several *mid-level managers* (at levels 2, 3...), and *operatives* at the lowest level [71]. In NSM, *managers* globally refer to all the top-level and mid-level managers, whereas operatives are called *agents*. Contrary to vertical delegation, we have *horizontal delegation* between two peers at the same level, typical of *cooperative paradigms* used in DAI. Distributed NSM may rely on a hierarchical paradigm, a cooperative paradigm, or a combination of the two. In fact, any paradigm outside the realm of centralized paradigms belongs to distributed NSM.

Delegation is normally a *one-to-one relationship*, between a manager and an agent in a hierarchical management paradigm, or between two peers (be they managers or agents) in a cooperative management paradigm. Arguably, delegation may also be considered, in some cases, as a *one-to-many relationship*, where a task is delegated to a group of entities, collectively responsible for the completion of the task. One-to-many delegation is forbidden by most authors in enterprise management [12, 71, 152, 241]. It can be considered in DAI though. In distributed NSM, we propose to classify it as a form of cooperation, by coupling hierarchical and cooperative paradigms: a manager delegates a task to an agent, and this agent in turn cooperates with a group of agents to achieve this task. In the case of a *many-to-many relationship*, we are clearly in the realm of cooperation rather than delegation.

2.1.7 Paradigm vs. technology

Some people confuse management paradigms with management technologies. A typical example is CORBA: in the literature, we find it referred to indistinctly as a paradigm, a technology, or even a framework. In the tradition of software engineering, and especially object-oriented analysis and design, we consider that technologies implement paradigms [82]. At the analysis phase, network and systems administrators select a *management paradigm* (e.g, distributed objects). At the design phase, they select a *management technology* (e.g., Java or CORBA). At the implementation phase, they use that technology to program the management application.

2.1.8 Architecture vs. framework

Many people confuse architectures with frameworks. In the IETF community, these two terms are often used interchangeably. For all SNMP protocols, the IETF uses the same name for the underlying protocol and the management architecture—unlike the International Organization for Standardization (ISO), for instance, which distinguishes the OSI management architecture from the communication protocol (CMIP, or Common Management Information Protocol) and the protocol primitives (CMIS, or Common Management Information Service). This has confused many people, especially when solutions were proposed to replace the SNMP

protocol while still preserving the SNMP management architecture. This problem has been acknowledged by the IETF, and a recent RFC [93] now recommends to refer explicitly to an *SNMP protocol* or an *SNMP management framework* (*SNMP framework* for short).

Unfortunately, frameworks and architectures have different meanings for the object-oriented community [70]. An *architecture* refers to the collection of models devised at the analysis and high-level design phases of an application; an architecture is therefore abstract in nature. A *framework*, conversely, refers to both an architecture and a set of template classes that implement this architecture. A framework provides hooks for site-specific extensions. It gives a common basis for different sites to build similar applications. It provides code that implements a specific model. A framework is related to a specific implementation and is language specific; an architecture is not. In this dissertation, we adopted the definitions of the object-oriented community. The three SNMP management frameworks are therefore called the *SNMP management architectures*, and the OSI management framework is called the *OSI management architecture*.

2.1.9 Integrated management vs. enterprise management

The meaning of *enterprise management* is also ambiguous. To most people, enterprise management is something you learn doing business studies; this is the sense we retain in this chapter. But a new fashion appeared in the mid-1990s, epitomized by Web-Based Enterprise Management (WBEM). Particularly in large, geographically dispersed corporations, the problem is not so much managing networks and systems *per se*: it is managing networks, systems, applications, services, and policies that are deeply intertwined. Before the days of WBEM, this was commonly referred to as *integrated management*. Since the first WBEM proposal (1996), it is often called *enterprise management*, or even *global enterprise management*. The main reason for this terminological change is that in the mid-1990s, the main focus of marketing people shifted from networks and systems to services, so they invented a new buzzword to epitomize this shift, but ignored the existing technical terminology. At IM'97, a speaker mentioned that the phrases *network management* and *systems management* had been "banned", so to say, in his company, in favor of *enterprise management*. This reflects the lack of substance behind this change of terminology. In our view, there is nothing more to this new so-called *enterprise management* than there is already in integrated management. As the latter is unambiguous, and because we see no reason beyond sheer marketing to change a well-accepted terminology, we will not use the expression *enterprise management* in its WBEM sense.

2.1.10 Firewalls

Real-life networks are becoming more and more different from the theoretical concept of a self-contained, connex network where the division between "us" and "them" is simplistic. Today, small enterprises often have remote offices and need somehow to manage them (at least the remote IP router). Midsize enterprises are typically geographically dispersed and have to manage remote branches. As for large enterprises, they must manage a collection of large networks, where the headquarters is interconnected to subsidiaries via private or public WAN links overlaid with a Virtual Private Network (VPN).

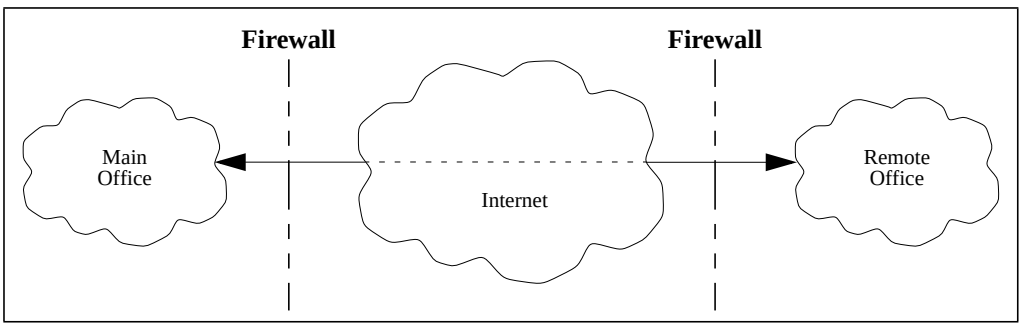

Fig. 2. Management across firewalls: the real picture

When the network is not connex, each constituent is usually protected by a firewall for security reasons. In this case, the manager at the main office accesses an agent at a remote office via one or several firewall(s). Real-life networks can thus be modeled as illustrated by Fig. 2. Each firewall depicted in this figure is optional.

The firewall of the main office is likely to be a full-blown, feature-rich, and expensive piece of software, whereas the firewall of the remote office is likely to be pretty minimalist (e.g., a filtering router).

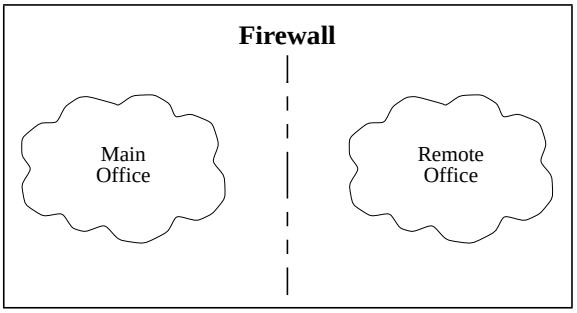

Fig. 3. Management across firewalls: the simplified picture

As far as manager-agent interactions are concerned, the scenario depicted in Fig. 2 can be abstracted as shown in Fig. 3. We will use this simplified representation in our dissertation.

2.1.11 Regular management vs. *ad hoc* management

The tasks achieved by a management application can be classified into four categories [132]:

- regular management
- *ad hoc* management
- configuration management
- background analysis

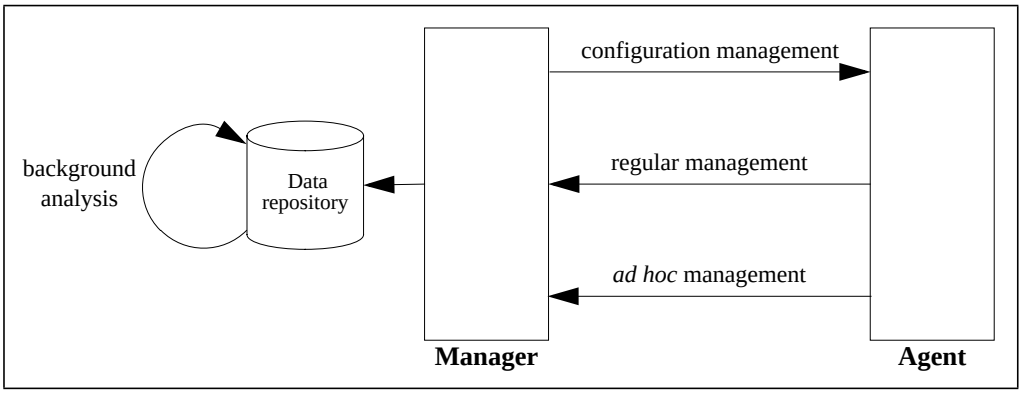

Fig. 4. Four categories of management tasks and their associated management-data flows

Regular management consists of management tasks that run continuously, almost permanently. It encompasses monitoring, data collection, notification delivery, and event handling. It requires that a dedicated manager be always up and running. In large networks, or in enterprises where the network is critical to the smooth running of the business, regular management is typically supervised by staff dedicated to monitoring (*operators*). Some Small or Midsize Enterprises (SMEs) cannot afford this and therefore do not perform regular management at all. Other SMEs rely on fully automated management (*unattended mode*): when a critical fault is detected by a monitoring application, the administrator is automatically paged; when a minor fault is detected, the administrator is simply sent an email. The data received by the manager for the purpose of *monitoring* is processed immediately, in pseudo real time, and then discarded. The data received for the purpose of *data collection* goes

directly to the data repository; it is not directly processed by the manager and serves for background analysis (see next). On the manager, the event correlator processes two types of events: some originating from the agents (*notifications*) and some originating from the manager itself (especially from its rule-based system). In regular management, the management-data flow goes from the agent to the manager (see Fig. 4).

Ad hoc management consists of management tasks that run only for a short time. It comprises troubleshooting and short-term monitoring. Troubleshooting is reactive; e.g., a network problem just showed up and an operator tries to identify and correct it by looking at error rates. Short-term monitoring is proactive; e.g., an administrator wants to check something on an IP router. By essence, *ad hoc* management cannot be automated. It is always manual and requires a user (administrator or operator) to interact with the management application. *Ad hoc* management takes place in virtually all companies. It complements regular management in large enterprises that can afford operators, or in SMEs that rely entirely on automated regular management. In small organizations, *ad hoc* management generally replaces regular management. There is no operator, no dedicated manager, and a part-time administrator who works in purely reactive mode, on an *ad hoc* basis. In *ad hoc* management, the management-data flow goes from the agent to the manager (see Fig. 4).

Configuration management is about changing the state of an agent to make it operate differently. It is active, as opposed to regular and *ad hoc* management that both passively collect data. We distinguish two modes: manual (e.g., with GUIs) and automated (e.g., with scripts). Manual configuration management usually takes place on an *ad hoc* basis, when the need arises. Automated configuration management is generally regular. For instance, the routing of transatlantic traffic over multiple links may be automatically altered every morning and every evening to benefit from slack-hour discounts. It can also occur on an *ad hoc* basis; e.g., when a new customer signs in, the type of service he/she pays for can result in the setting of access-control lists at an access router. Note that for configuration management, the management-data flow goes in the opposite direction, from the manager to the agent (see Fig. 4). This is why we do not split configuration management into regular and *ad hoc* management.

Background analysis includes report generation (typically on a daily, weekly, and monthly basis), billing, security analysis, data mining, and more generally any application that exploits and makes sense out of the wealth of data gathered by regular management, especially by data collection. These applications run offline, in the background, on a machine which is often not the manager. This prevents number-crunching background tasks from slowing down monitoring and event correlation, which are supposed to operate in pseudo real time. There is no management-data flow associated with background analysis between the agent and the manager (see Fig. 4).

In this dissertation, we will concentrate on agent-manager and manager-manager interactions, that is, on the organizational and communication models (see Section 6.1). We will mostly study regular management, which accounts for most of the network overhead, manager's processing overhead, and agent's processing overhead. We will also briefly consider *ad hoc* management.

2.2 Characteristics of SNMP-Based Management

Now that we have defined a clear terminology, let us summarize the main characteristics of SNMP-based management.

Partial specification

Many characteristics of SNMP-based management appear explicitly in the specifications. But some are transmitted more informally via well-known textbooks, research articles, magazines, SNMP-related mailing lists, or open-source software. For instance, the change in the recommended use for the `inform` Packet Data Unit (PDU), from a manager-to-manager notification in SNMPv2 to a general-purpose acknowledged notification in SNMPv3 [125, 167], does not explicitly appear in any RFC, but gradually imposed itself in the mailing list of the IETF SNMPv3 Working Group. (Note that we are not saying here that the new `inform` is

not specified.) Another example will be encountered in the next chapter, when we study the issue of the maximum size of an SNMP message: one value (1472 octets) is considered well-known only because it appears in one of the most famous textbooks on SNMP and was used in some open-source implementations of SNMP in the early 1990s. Note that we will also come across significant characteristics of SNMP-based management that have no technical roots at all and are solely due to the way the SNMP market developed over time.

Interoperability

SNMPv1 was developed in the late 1980s, in the days of proprietary management. Just like its main competitor in those days, the OSI management architecture, the main goal of the SNMP management architecture was to achieve interoperability between multiple vendors, to pave the way from proprietary to open management. Interoperability was undeniably the main merit of SNMPv1.

In the SNMP management architecture, interoperability is guaranteed by:

- SNMP agents and managers complying with the IETF specifications
- well-known generic MIBs evolving slowly
- vendor-specific MIBs whose internals are advertised
- a single metamodel for all MIBs
- a *lingua franca* (SNMPv1) used by most sites, and a small variation of it (SNMPv2c) used by the others
- management platforms developed by third-party vendors with supposedly no interest in favoring one equipment vendor over another

Simplicity

OSI management did not encounter any success in the IP world because it was overly complex. It tried to do everything right, and failed. SNMP, conversely, focused on the bare minimum, the bottom line, and postponed everything else *sine die*: security, distribution, etc. As a result, SNMP was simple and inexpensive to implement for equipment vendors. Open-source code for SNMP agents was freely available on the Internet (the Carnegie Mellon University (CMU) and Massachusetts Institute of Technology (MIT) distributions were probably the most popular in the early 1990s, when the SNMP market exploded), so it was very easy for a vendor to take this code, customize it, and integrate it in a piece of equipment. SNMP was simple to learn for customers, too, and it took little time for administrators and operators to get accustomed to it. Simplicity was the second main achievement of SNMP after interoperability and, in our view, the main reason for its success.

Generic vs. vendor-specific MIBs

Agents support two types of virtual management-data repositories (MIBs) in the SNMP world. Generic MIBs are usually defined by the IETF[1] and supported by multiple vendors (e.g., MIB-II). Vendor-specific MIBs are controlled by the vendors but available to all (e.g., the Cisco MIB, initially unique and now broken down into several entities). MIBs constitute the main element of the SNMP information model.

A stable metamodel[2]: SMI

Another characteristic (and strength) of SNMP is that its information metamodel, the Structure of Management Information (SMI), has been very stable over time. There have been only two versions of it in a decade. SMIv1 [177] is still used by MIB-II, *the* standard MIB in the IP world, supported by virtually all SNMP-compliant network devices and systems. Most other MIBs now use SMIv2 [44].

---

1. Some MIBs are defined by other consortia; e.g., the SNMP M4 Network Element View MIB is defined by the ATM Forum [14].
2. *Stricto sensu*, SMI is not SNMP's information metamodel, but rather the language used for defining SNMP's information model (MIBs). The IETF did not specify a metamodel *per se*, if we refer to the exact definition of a metamodel [13]. But the vocabulary of the SMI language (that is, the *ontology* in modeling jargon) relies on an implicit metamodel that we also call SMI in our dissertation.

For completeness, we should mention that a third version of SMI, called SMIng, was recently proposed by Schönwälder and Strauss [183]. But it is still confined to the research community, and there is no evidence as yet that it will be standardized by the IETF, let alone adopted by the SNMP market. We therefore do not consider it when we refer to SNMP-based management.

Not one but several SNMPs

When we say *SNMP*, we actually make a simplification. There are currently three SNMP management architectures (SNMPv1, SNMPv2c, and SNMPv3), three SNMP protocols (SNMPv1, SNMPv2, and SNMPv3), two metamodels (SMIv1 and SMIv2), and many Protocol Data Units (PDUs): `get`, `get-next`, `get-bulk`, `set`, `trap`, `SNMPv2-trap`, `inform`, etc. Each management architecture uses a different version of the protocol. Different protocols accept different PDUs. For instance, as far as notifications are concerned, the SNMPv1 protocol uses the `trap` PDU, whereas the SNMPv2c and SNMPv3 protocols use the `SNMPv2-trap` PDU. All of this is slightly confusing because the clear separation between the versioning schemes of PDUs, protocols, and management architectures was only made in 1998, with the first release of SNMPv3. We will come back to this in more detail in Section 3.1.

Manager-agent paradigm, client-server architecture

As OSI management, SNMP-based management relies on the well-known manager-agent paradigm [162]. This paradigm is based on the client-server architecture for communication. For polling, the manager is the client and the agent is the server. For notifications, it is the opposite. In hierarchically distributed management, the same machine can play the manager and agent roles; in this case, it runs a client and a server.

All agents are created equal... and "dumb"

In the SNMP management architecture, and especially in its organizational model, there is an implicit assumption that all agents should be managed alike. As a result, we use the same type of interaction between a manager and an agent, whatever the agent, be it powerful or not. In such situations, the weakest link of the chain rules: all agents must be managed alike, and some agents can be "dumb"[1], so all agents are managed as if they were "dumb". The manager does not have to cope with "dumb" agents and "smart" agents in parallel: there is no differentiation. As a result, SNMP leverages the least common denominator between bottom-of-the-range and top-of-the-range equipment.

The suitability of this implicit assumption was destroyed in the mid-1990s by Wellens and Auerbach, when they exposed the *Myth of the Dumb Agent* [242], and by Goldszmidt, when he justified the suitability of his new organizational model: Management by Delegation [88]. The IETF recently became aware of this and the Script MIB [126], released in 1999, is its first achievement to provide management *à la carte*, with a differentiation by the manager between "smart" and "dumb" agents. But for security reasons, the Script MIB requires SNMPv3, which is hardly used anywhere at the time we write these lines.

Small footprint on agents

Because agents are supposedly "dumb" and short of resources, the footprint of SNMP on agents is very small—as opposed to OSI management, for instance, which imposes a large footprint on agents. Apart from the MIBs that they support, all the network devices and systems of a given vendor can therefore run the same SNMP code, from bottom- to top-of-the-range equipment.

1. This term is often used in NSM: we have "dumb" network devices or systems just as we have "dumb" terminals. "Dumb" agents do not really participate in the management application, they only passively collect data on behalf of the manager.

Monitoring is not notification driven

In the telecom world, most of the problems with the agents are detected by the agents themselves. When a problem is encountered, the agent generates a notification and sends it to the manager. Parallel to this, the manager's rule-based system routinely checks a number of things (e.g., that a certain aggregate value remains below a certain threshold), and generates an event when a problem is detected (e.g., a threshold is exceeded). The main task of the manager, with respect to monitoring, is to correlate all these types of events: those generated by the agents and those generated by the manager. Because most events originate from the agents, we say that monitoring is notification driven in the telecom world.

In the IP world, conversely, most of the problems with the agents are detected by the manager. This is a direct consequence of the assumption that the agents are "dumb" and short of resources: they are not sufficiently instrumented to detect what goes wrong. Most of the agents in the IP world are only able to send a handful of notifications to the manager. For instance, with IOS 10.3[1], Cisco IP routers were only able to send 7 types of notifications in 1996. Reportedly, this was still the case in March 2000 (IOS 12.0), although this information could not be verified by the author on real equipment. During his previous job, the author came across some pieces of equipment that could not send any notification. Because few events originate from the agents, we say that monitoring is not notification driven in the IP world. This mode of operation explains why notifications are not very important in the IP world. To detect problems, a manager typically relies on polling instead of notifications (see next). Note, however, that nothing inherent in the SNMP specifications prevents vendors from basing monitoring on notifications issued by agents.

Polling for regular management

In SNMP, monitoring and data collection are based on *polling* (pull model). The manager patiently keeps asking the same things to the agents, at each poll cycle, and the agents return the values of the requested MIB variables. On the manager, a module that we call the *polled-data interpreter* is in charge of detecting whether anything goes wrong with the agents. It does so by "interpreting" (analyzing) the data received from the agents.

The SNMP designers initially promoted a concept known as *trap-directed polling* [176, p. 20, 207, pp. 79–81]: upon receipt of a notification from an agent, a manager was expected to poll a number of MIB variables on this agent to work out the cause of the reported problem. This vision has been abandoned in practice, because agents are typically not instrumented to generate many different notifications. To learn that an agent experiences a problem, a manager must actively poll the agent and cannot passively wait for incoming notifications.

Vendor-specific management GUIs

Management platforms generally come with a few generic management GUIs, but most customers complement them with vendor-specific management GUIs called *add-ons* (e.g., CiscoWorks for Cisco equipment). These add-ons are dedicated to one specific vendor or one line of products (e.g., IP routers or ATM switches). They offer user-friendly GUIs, customized for a specific piece of equipment. For instance, to report that a port has gone down on a network device, a vendor-specific GUI typically displays a picture of the connectors at the back of the device (the image looks exactly like the real device), and turns the corresponding port red; this makes it easy for an operator to check that the corresponding connector is not loose. A generic GUI, conversely, simply represents a generic device with ports randomly placed; this view does not look like the real device, and finding the actual location of the corresponding port on the device can take time.

Add-ons are commonly used today in Network Operations Centers (NOCs) in the IP world, because they make the life of operators a lot easier. They have an important drawback, though: they defeat the purpose of open management. Network devices and systems are supposed to be commodities: if vendor A is less expensive than vendor B, the customer buys equipment from vendor A. Some time later, if vendor B is less expensive, the customer purchases from vendor B. With vendor-specific GUIs, changing a supplier involves retraining

1. Internetwork Operating System (IOS) is the operating system run by Cisco's networking equipment.

operators—something companies legitimately want to avoid. Differences between command-line interfaces from different vendors make this issue of retraining even worse. With these discrepancies, equipment vendors managed to segment a large commodity market into a collection of captive markets, which are obviously more lucrative [132].

No distribution between managers

The SNMP management architecture is centralized. There is currently no standard and open way of distributing management across a hierarchy of managers in SNMP-based management, especially when the managers are written by different vendors. This issue will be studied in Section 3.1. To work around this limitation, some vendors support distributed management through proprietary extensions. A few years ago, for instance, HP OpenView offered an OSI-based solution for distributing management in the IP world; all the managers had to run HP OpenView.

One consequence of this lack of support for open distributed management is that problems that occur on management-domain boundaries are generally difficult to analyze, and are often dealt with by people on an *ad hoc* basis. Another problem is that it is very difficult to integrate the management of two domains when the managers come from different vendors. This situation typically arises when two companies merge.

Most management platforms support only a fraction of FCAPS

In the telecom world, management platforms generally support most (if not all) of the OSI functional areas (Fault, Configuration, Accounting, Performance, and Security management, also known as FCAPS [48, 49]). 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[1].

The *mandatory tasks* are performed by virtually all management platforms. They are:

- Monitoring for the purpose of reactive fault management, reactive performance management, and reactive provisioning.
- Data collection for the purpose of proactive fault management, proactive performance management, and proactive provisioning.
- Data interpretation (because agents cannot work out the origin of problems by themselves).
- Event correlation. Events can be SNMPv1 traps (`trap` PDU), SNMPv2 notifications (`SNMPv2-trap` PDU), SNMPv3 informs (`inform` PDU), or events generated by the monitoring engine.

We call *optional tasks* all the remaining management tasks: configuration management, inventory management, access-control management, accounting, billing, security-logs analysis, etc.

MIBs offer instrumentation APIs

Due to the way the SNMP market developed over time, SNMP MIBs offer only low-level APIs, often called *instrumentation MIBs*. These MIBs deal only with the nuts and bolts of NSM. This is not inherent in the SNMP management architecture itself, nor in its information model: it is simply historical. Note that the work on high-level virtual management-data repositories recently undertaken by the Distributed Management Task Force (DMTF) and the IETF, under the umbrella of policies and policy-driven management, might eventually bring an end to this lack of high-level MIBs in SNMP.

1. These different cultures partly explain the different approaches to billing between traditional telephony and IP telephony, for example. In the IP world, accounting is also particularly difficult to manage when we use dynamic IP-address allocation schemes such as the Dynamic Host Configuration Protocol (DHCP).

New MIBs appear at a slow pace

In the IETF working groups, MIBs are created and modified by consensus. This is at the heart of the IETF's business model. Reaching consensus takes time, so new MIBs appear at a slow pace. As time goes by, the commercial interests behind any technical decision become increasingly important, and more and more vendors get involved in the IETF working groups with both technical and commercial agendas. Thus the decisions take even longer to be made. The slow pace at which policy-based management has progressed since it is reasonably well understood (that is, since the mid-1990s [196, 244, 5]) illustrates this phenomenon.

Tightly coupled data repository

For commercial rather than technical reasons, customers in the IP world depend on peer-to-peer agreements between SNMP-based management-platform vendors and third-party database vendors. This is due to the tight coupling of the data repository and the management platform in this market.

If a customer already owns and masters a Relational DataBase Management System (RDBMS), and then buys a management platform[1], he/she cannot necessarily use this RDBMS to store management data: this database must be already supported by the management platform of his/her choice. Since only a fraction of the databases on the market are supported by the major management platforms, he/she has to be lucky... Alternatively, he/she can be charged the port of the management platform to this new RDBMS—but this cost is prohibitive for most companies. This uncomfortable situation is caused by the way management-platform developers usually write their code and interface with data repositories in the SNMP world: they use the proprietary APIs offered by RDBMS vendors (e.g., the C-language SQL interface to Oracle) to get the best performance from the RDBMS. While this renders database accesses efficient, it also makes it costly to support multiple RDBMSs.

No integrated management

The last important characteristic of SNMP-based management is the absence of integration of network, systems, application, service, and policy management. This is not due to intrinsic limitations in the SNMP management architecture, but rather to the lack of interest (or at least, the lack of deliverables) in integrated management within the IETF working groups. As a result, enterprises run different management platforms from different vendors in parallel and independently. Today, there is no standard and open way of integrating management in the IP world. The few management platforms that claim to support integrated management to a certain extent (e.g., HP OpenView and CA Unicenter TNG) do so by using proprietary solutions.

The current race to standardize and implement policy-based management, and to translate high-level policies into low-level settings of NSM MIB variables, has left the IETF standing in the starting-blocks for a while. This situation created an opportunity for a new industrial symposium, the DMTF, to take the lead in integrating network, systems, application, service, and policy management in the IP world. We will come back to the DMTF work in more detail in Section 5.5.1.

2.3 Strengths of SNMP-Based Management

Before we set about criticizing the SNMP management architecture, the SNMP protocol, and SNMP-based management platforms, let us first acknowledge the tremendous success experienced by SNMP (especially SNMPv1) in the 1990s. Within a few years, the networking industry, which was entirely dominated by proprietary equipment and management, turned to open systems and open management. IP networks have undoubtedly been the greatest success of open systems.

1. This situation was very frequent in the first half of the 1990s, as RDBMSs predated SNMP-based management platforms.

As we reviewed the main characteristics of SNMP-based management in the previous section, we came across the following strengths of SNMP-based management:

- interoperability
- simplicity
- wide support by IP-equipment vendors
- small footprint on agents
- low extra cost of adding management to a network device or system
- same minimal middleware everywhere

In light of the technologies widely used today, many design decisions in SNMP-based management appear inefficient or outdated, as we mentioned in the introduction. But they did not in the late 1980s, when SNMPv1 was devised. Moreover, if we place ourselves in a historical perspective taking into account how the management market evolved [132], many deficiencies in today's commercial management platforms can be analyzed and understood. The success of SNMP-based management, especially in network management, is due to a large extent to its simplicity, so it would be unfair to criticize this simplicity afterward. In short, congratulations to the designers of SNMP! They did a great job in the context of proprietary management that characterized the late 1980s. Turning a closed, proprietary market into an open, standard-based market within a few years is a remarkable accomplishment.

2.4 Problems with SNMP-Based Management[1]

During a decade of operational use in the IP world, SNMP-based management has exposed a number of problems. Some of them are minor, but others are serious and require drastic changes in the management architecture, communication protocol, or management platforms. In this section, we describe the main problems that we identified and, for each of them, briefly summarize the solution offered by WIMA. We first study the problems related to scalability and efficiency, then the problems related to management platforms, and finally identify some important features that are currently missing in SNMP.

2.4.1 Scalability and efficiency issues

Throughout the 1990s, several independent evolutions gradually exposed a major weakness in the SNMP management architecture: scalability. We will see that the efficiency problems experienced by the SNMP protocol have a strong impact on the scalability of the SNMP management architecture.

Scalability issues can be classified into four categories:

- network overhead
- latency
- manager's processing capacity
- capacity of the manager's local segment

The first factor that impacts scalability is 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 the lower the network overhead the better. In general, network overhead should not account for more than a small percentage of the network capacity—except for the manager's local segment, as we will see. The traffic associated with NSM is considered *entirely* as network overhead, so an important goal in NSM is to keep network overhead low. For a given capacity,

1. A small fraction of the material presented in this section was published in March 1999 in *The Simple Times*, the online magazine of the SNMP community [201]. Ron Sprenkels coauthored this article and contributed some ideas presented here; he also invented the expression `get-bulk` *overshoot effect*. Several ideas also came up during the first meeting of the IRTF Network Management Research Group in Lausanne, Switzerland in November 1998.

network overhead increases almost linearly with the amount of management data that is transferred (*stricto sensu*, it is a non-continuous piecewise linear function due to the IP, transport, and application headers).

The second factor is 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. If it is too high, the manager can reach a point where it does not fulfill its mission: it detects operational problems too slowly and corrects them too late, thereby possibly causing new problems. Latency can be split into two components. *End-host latency* includes marshalling and unmarshalling of data, compression and decompression, security-key computation, etc. It mostly depends on the effectiveness of the communication protocol, and we will see that the SNMP protocol leaves a lot to be desired in this respect. Keeping end-host latency low on the manager frees up some resources that can be allocated to the actual management application. Keeping it low on the agent prevents the management processing overhead from hampering the operation of the agent (the purpose of an agent is to fulfill its mission, not to be managed). *Network latency* includes the time spent in the network links (propagation delay) and network equipment. It depends on the capacity and error rates of the links, on the speed of the IP routers traversed between the agent and the manager (software- or hardware-based routing, busy or empty input-output queues), on the speed of the switching fabric, etc. The main effect of the management application on network latency is the amount of data to move about. This impact varies as the network overhead. The main effect of the communication protocol on network latency is the number of messages that are exchanged (see Section 2.4.1.2).

The third factor with a direct impact on scalability is the manager's processing capacity. The manager's hardware resources (CPU, memory, etc.) that are dedicated to the management application cannot be indefinitely increased, due to cost and hardware constraints; so there is only so much that the manager can process per time unit. In Section 3.1.1, we will explain that the three SNMP management architectures (v1, v2c, and v3) follow a centralized paradigm, despite some (failed) past attempts to distribute management. This centralization exacerbates the need to relieve the manager from performing any superfluous processing.

The fourth factor is the capacity of the manager's local segment. This is a direct consequence of centralized management. The management data sent by all the agents converges toward a single point: the network segment to which the manager is connected to. By design, this creates a bottleneck. It is therefore important to reduce the network overhead as much as possible, especially by improving the efficiency of the communication protocol, in order to postpone the threshold at which the manager's local segment saturates and cannot cope anymore with all the incoming data. Note that for large or busy networks, the manager's local segment is usually dedicated to management; in this case, network overhead is not limited to just a small percentage: it is limited solely by the segment capacity.

2.4.1.1 Ever more management data

Over time, scalability has become a serious issue in NSM. This is because the total amount of management data that must be transferred over the network, and processed by the manager, has gone up very significantly and keeps growing. Despite the increase in the installed network capacity (by about an order of magnitude for LANs during the 1990s), the network overhead caused by management data is larger than ten years ago, and is expected to continue to grow. We identified three core reasons for this need to transfer and process ever more management data.

First, the number of agents to manage has exploded, because:

- The installed base of IP network devices and systems has increased dramatically throughout the 1990s, by several orders of magnitude. Most of the growth was fueled by Personal Computers (PCs) until the late 1990s. Mobile phones and Web-enabled handheld devices are now spreading even faster.
- The proportion of equipment supporting TCP/IP has grown very rapidly. Today, the TCP/IP stack is virtually ubiquitous. In contrast, most PCs did not have a TCP/IP stack when SNMPv1 was released.
- The proportion of networked equipment has grown asymptotically toward 100% in the 1990s, whereas many machines (especially PCs) were stand-alone in the late 1980s.

Second, the total amount of management data that needs to be retrieved per agent has gone up significantly, because:

- Agents support more MIBs today than in the early 1990s. The days are over when an IP router supported only MIB-II and a vendor-specific MIB.
- The size of many SNMP tables has increased with the size of the networks (e.g., IP routing tables, TCP connection tables, and accounting tables).
- The number of ports per interconnection device has increased by an order of magnitude between the typical equipment of the early 1990s (routers, bridges, and repeaters) and the typical equipment of the late 1990s (intelligent hubs, level-2 switches, and level-3 switches).
- A growing proportion of enterprises critically depend on their networks for the smooth running of their business. As a result, they follow more closely than ever the health of their networking equipment (smart proactive management with management platforms, as opposed to simple reactive management performed manually).
- The overprovisioning of networks, which has imposed itself over the years as an absolute necessity to network architects, is costing more and more as networks grow in size and capacity. Regular and detailed performance monitoring helps reduce the overprovisioning ratio and allows for substantial savings.

Third, the amount of management data to move about and process is expected to continue to increase in the future, because:

- The installed base of IP network devices and systems keeps growing, year after year.
- The advent of multimedia networks and pseudo real-time services (e.g., streaming video) calls for QoS management, and thus more transfers of management data.
- Service management, which only recently appeared in the IP world, generates extra management data.
- If SNMPv3 is used in the future, some of its most useful new features (authentication and access control) may require large configuration tables to be transferred over the network, especially View-based Access Control Model (VACM) tables [246].
- If *ad hoc* networks become a reality, the number of agents to manage will explode (but each agent, taken individually, will require little management).

SNMP-based management cannot cope indefinitely with this increase in the amount of data transferred over the network and processed by the manager. As a matter of fact, SNMP is exposed to the four scalability problems described on p. 21. To keep network overhead low on each network link, there is a limit to the amount of management data that each agent can send. To keep the manager's latency low, there is a limit on the total amount of data that the manager can receive from all the agents. To ensure that the manager keeps up with the workload of the management application without falling behind, there is a maximum amount of data that can be processed per time unit. And finally, to keep the load on the manager's local segment below its saturation point, there is a limit on the total amount of data that can be sent by the agents to the manager. As the amount of management data grows indefinitely, sooner or later, we must reach one of these limits: it is not possible, due to cost and hardware constraints, to increase indefinitely the capacity of all the network links and the manager's processing capacity. SNMP-based management has no built-in mechanisms to avoid these threshold effects.

In WIMA, we face the same increase in the amount of management data to transfer and process; this is a given for all management architectures. We deal with it in two ways. First, we improve the efficiency of data transfers by changing the communication and organizational models. Once we reach one of the limits mentioned above, we split the management domain and distribute management across several managers.

2.4.1.2 Bulk transfers: too many SNMP messages

As the amount of data to transfer grows, it makes sense to reduce the overhead by sending the data in bulk, that is, by sending many MIB variables at a time (typically, between 10 kbytes and 1 Mbyte of data). The problem is that SNMP is not suited for this. In SNMPv1, v2c, and v3, the retrieval of several MIB variables at a time requires the exchange of many request-response PDUs over the network. We identified three basic problems: the design of `get-next`, the `get-bulk` overshoot effect, and the maximum size of an SNMP message.

Design of `get-next`

With SNMPv1, we have only one practical option to retrieve a table: the `get-next` protocol operation. The problem with bulk transfers is that, by design, `get-next` cannot fill up SNMP messages efficiently. If the table has many rows, the manager must perform at least one `get-next` per row. If a table row does not fit entirely into a single SNMP message, the agent returns an error message; the manager then has to issue a new `get-next` with a new varbind list[1] consisting of only some of the OIDs in the table row. This guess-work by the manager ("How much space will the agent require to answer my request?") is inefficient. First, it unnecessarily increases the number of computations by the manager and therefore increases end-host latency. Second, because the manager fills up the messages by trial and error, the filling ratio can be poor. By putting too little data, the manager unduly increases the number of round trips, which increases the network latency, the network overhead, and the end-host latency. By putting too much data, the manager causes an error at the agent, which unnecessarily augments network overhead, network latency, and end-host latency at the manager and the agent.

Sprenkels's `get-bulk` overshoot effect

With SNMPv2c and SNMPv3, we can also use the `get-bulk` protocol operation to retrieve a table. With `get-bulk`, we can retrieve more than one table row at a time. If a table row does not fill up an entire SNMP message, `get-bulk` is more efficient than `get-next` because we can transfer more data per SNMP message. This reduces the network overhead, the network latency, and the end-host latency. But the manager still has to perform some guess-work because of the maximum size of an SNMP message; Sprenkels calls this problem the *`get-bulk` overshoot effect* [201].

Because the manager does not know in advance the length of the table it wants to retrieve, it has to guess a value to use for the *`max-repetitions`* parameter. Using a low value causes more PDU exchanges than necessary. Using a high value, however, can result in an *overshoot* effect: the agent can return data that does not belong to the table of interest to the manager. For instance, let us suppose that `max-repetitions` is equal to 50 and the table only contains 10 additional rows. The agent will first receive the request, unmarshall it, and identify the input varbind list. Then, 50 times in a row, it will retrieve object values from the SNMP instrumentation, encode them with Basic Encoding Rules (BER), and store them in a large buffer in memory. Finally, it will send the SNMP message to the manager. The manager will have to decode the 50 objects in the varbind list returned by the agent, but will keep 10 and discard 40. This is a waste of network resources and a waste of processing resources at both the manager and the agent: many OIDs are BER-encoded, marshalled, transferred, unmarshalled, and BER-decoded only to be disposed of by the management application.

Scotty [187] and some other network management applications implement a slow-start adaptive mechanism to find out empirically a good value for `max-repetitions`. This mechanism only partially addresses the issue, though. First, if the initial seed for `max-repetitions` is too large, we still experience an overshoot effect. In the previous example, if the seed is set to 20 instead of 50 due to slow start, we process 10 useless objects instead of 40—better, but still not very good. Second, if the initial seed for `max-repetitions` is too small, this mechanism only partially fills the first SNMP messages and more messages are sent than really necessary—again, not optimal.

---

1. A *varbind list* is a vector of MIB variables in SNMP (*varbind* stands for *variable binding*). An OID (Object IDentifier) uniquely identifies a MIB variable.

Maximum size of an SNMP message

The third problem is the maximum size of an SNMP message: it is too low for bulk transfers. This size is always constrained by the following factors:

- We must have exactly one PDU per SNMP message [45].
- An SNMP message cannot expand over multiple UDP datagrams, by design of UDP (packetization is performed by a layer on top of the transport protocol).
- UDP datagrams cannot be longer than 64 kbytes [169].
- All SNMP agents must accept SNMP messages that are up to 484 octets in length, but may legally refuse longer messages [46, 239].
- Some TCP/IP stacks (especially old ones) limit the maximum size of a UDP datagram to 8 kbytes. This is a carry-over from the days when memory was a scarce and expensive resource on agents[1].
- SNMPv2c defines the `maximum message size` [45], which allows the manager to further constrain the maximum size of an SNMP message generated by the agent. The problem is that SNMPv2c does not convey this information from the manager to the agent, thus it is unusable in practice. SNMPv3 solved this problem by clearly specifying how to transfer the `maximum message size`.

The maximum size of an SNMP message can also be constrained by the following optional factors:

- The size of a UDP datagram is limited to the Maximum Transfer Unit (MTU) if we must not fragment at the IP layer. This typically occurs on busy backbone routers, which are generally configured not to fragment IP packets so as to conserve resources for routing. This MTU is typically equal to 1472 bytes for a LAN (1500 bytes of Ethernet payload, less 20 bytes of IP header and 8 bytes of UDP header) and 548 bytes for a WAN (576-byte MTU for nonlocal destination network [31], less 20 bytes of IP header and 8 bytes of UDP header).
- In his famous textbook [176], Rose specifies minimum guaranteed sizes for SNMP messages for each transport domain. These values are: 1472 octets for UDP [176, p. 171], 1472 octets for the OSI ConnectionLess Network Service (CLNS) and Connection-Oriented Network Service (CONS) [176, p. 172], 484 octets for Appletalk's DDP [176, p. 173], and 576 octets for Novell's IPX [176, p. 174]. Although these values are not specified in any RFC, they reflect habits in the SNMP market. Note that the recommended value of 1472 octets for UDP is identical to the LAN MTU in the previous bullet.
- A `get-next` request cannot retrieve more than one table row at a time.

In short, the maximum size of an SNMP message is always comprised between 484 bytes and 64 kbytes. Typically, it is equal to 1472 bytes for a LAN and 548 bytes for a WAN. Old SNMP agents have a maximum size of only 484 bytes. Let us translate this in terms of SNMP messages exchanged over the network. When the manager retrieves several MIB variables or tables in one bulk, the agent must split the data into many SNMP messages: 10 kbytes divided by 1472 yields only 7 SNMP messages, but 1 Mbyte divided by 484 requires more than 2000 SNMP messages! If the table has many rows and each individual row is not particularly large, the SNMP messages are only partially filled, so their number is even greater when `get-next` is used. (This is commonly the case in production networks today, because they use SNMPv1 in their majority).

This multiplication of SNMP messages is highly undesirable because the overall latency of a data transfer increases with the number of PDU exchanges. The reasons for this are fourfold. First, end-host latency increases because both the agent and the manager have to process more packets, generate or parse more headers, marshall or unmarshall more data, etc. Second, network latency augments because more SNMP messages induce more round trips and increase the overall transmission delay[2]. Third, a larger number of

1. We find a trace of this limitation in the Network File System (NFS), another application-layer protocol based on UDP. In 1989, when NFS version 2 was specified, the maximum size for data transfers was set to 8 kbytes [215, p. 21]. This limit was relaxed only in 1995, when NFS version 3 was released [41, pp. 11–12].
2. Note that in some environments, e.g. Ethernet-based LANs, many small packets actually have a better chance to go through without retransmissions than fewer large packets, so network latency may not be badly affected in such networks.

SNMP messages has a slightly adverse effect on network overhead because the higher the number of packets, the higher the number of headers to move about and the higher the network overhead. Last but not least, this multiplication has an impact on routers along the path between the agent and the manager, because what takes time in a router is the header parsing and analysis; once the router knows which interface to send the packet to, what the Type Of Service (TOS) field is, and what the IP options are, the length of the packet has comparatively little impact on the latency. Therefore, the more packets the larger processing overhead for all traversed IP routers. By unnecessarily increasing the number of packets, we unnecessarily fill up the buffers used by the IP routers for input and output queues, and therefore increase the risks of having buffer overflows.

In WIMA, we address the issue of bulk transfers by allowing for indefinitely large messages to be exchanged between the agent and the manager (or between the mid- and top-level managers in distributed hierarchical management).

2.4.1.3 Table Retrievals

In the previous section, we described how cumbersome it is to retrieve SNMP tables with `get-next` and `get-bulk`. Two other problems make this situation even worse in all versions of SNMP (v1, v2c, and v3): the possible presence of holes in tables and the consistency of large tables.

Holes in sparse tables

Retrieving SNMP tables is more complicated than retrieving simple MIB variables (managed objects) because the SNMP management architectures do not define tables *per se*, in the common sense of the term; instead, they define *conceptual tables*. In a conceptual table, none of the columnar objects is mandatory in a table row; in other words, conceptual tables allow their rows to have missing cells that we call *holes*. In the presence of holes, tables are called *sparse tables*. Let us illustrate this problem with an example.

With SNMPv1, it is common practice to retrieve a table from an agent by performing repeated `get-next` operations at the manager. The retrieval process starts off with a varbind list initialized with the OIDs of the first row of the table. Afterward, the varbind list of each `get-next` operation holds the last retrieved OID in each column. If a hole is encountered in table row N (that is, the corresponding cell of the table is not defined by the vendor for that device), the `get-next` operation returns the next object that does have a value in that column. The row of that object is *a priori* unknown: it could be row (N+1), (N+2), etc. The manager detects a hole by comparing, column by column, the expected next OID with the OID returned by `get-next`. If they differ, the next expected OID is a hole. All this processing increases the end-host latency at the manager when the table is large, be the table sparse or not.

If none of the remaining table rows has a value in that column, then the next OID in the MIB is returned (the rule for MIBs traversal is to use a depth-first tree search). The retrieval ends when all OIDs in the returned varbind list start with a sequence which is different from that of the table; e.g., for the IP routing table of MIB-II, when none of the OIDs starts with 1.3.6.1.2.1.4.21. Thus, the overshoot effect described earlier for `get-bulk` is integrated *by design* in the retrieval of conceptual tables, which is remarkably inefficient.

In retrospect, one wonders why a clean concept of table was not introduced from the outset in SNMP. Would it not be more intuitive to return the error code `noSuchInstance` when an empty cell is retrieved from a table? We identified four potential reasons justifying this design decision.

First, this scheme was devised by the designers of SNMPv1 in the late 1980s, when memory was a scarce and expensive resource in agents. The holes in sparse tables allowed vendors to save memory in the agents. Later, while agents were fitted with more memory, conceptual tables remained unchanged in SNMPv2c and SNMPv3 for backward compatibility. Probably all agents put on the market today have enough memory to store an error code in memory when a cell is empty, hence the argument of the scarcity of memory does not hold anymore.

Second, in those days, it seemed a good idea to keep network transfers as slim as possible; every single byte that could be saved had to be saved. By explicitly allowing the agent to skip holes in a table [178, pp. 3–4], the

designers of SNMPv1 helped keep the network overhead low and redeemed the verbosity of BER-encoded data (see Section 2.4.1.4). Today, the speed of development of an application generally prevails over the network efficiency of the solution (there are exceptions, though). Layered approaches such as CORBA or Distributed Component Object Model (DCOM) are routinely used to develop software, despite the fact that they induce a large network overhead and do little to reduce it. Therefore, it no longer makes sense to save every single byte in SNMP when most of the user traffic is grossly inefficient in terms of network overhead.

Third, when SNMPv1 was designed, the manager was supposed to have plenty of resources (as opposed to the agent), so the extra processing induced by this convoluted encoding of tables was not deemed to be a problem. Today, networks are much larger than in the early 1980s and managers often get close to (and sometimes reach) the limits of their resources. Putting the unnecessary burden of reconstructing weirdly structured tables on the shoulders of managers is no longer considered a good idea.

Fourth, there was no `noSuchInstance` error code in SNMPv1: this code was only introduced in SNMPv2, the precursor of SNMPv2c, and later maintained in SNMPv3.

In summary, holes in tables were not a problem when SNMPv1 was designed; but they are today, because SNMP-based management relies heavily on OIDs stored in tables and the decoding of these OIDs by the manager is inefficient and increases the latency. We need a better mechanism for table retrievals.

Consistency of large tables

A potentially serious, yet often forgotten, problem encountered with SNMP tables is their consistency. With all existing SNMP protocols (v1, v2, and v3), the manager has no way of retrieving an entire table in a consistent state. If the agent updates the table in the middle of a retrieval, the manager ends up with an inconsistent view of this table. This issue is particularly critical for large tables, whose retrievals demand the exchange of many messages over the network, and for which the elapsed time between the first request and the last response can exceed the mean time between two consecutive changes of the table.

The retrieval of an entire MIB in a consistent state can reasonably be considered unrealistic: this might require that the agent halt its operation for several seconds, which, in many cases, is neither possible nor desirable. But retrieving an entire table in an atomic operation is a very reasonable expectation from administrators.

The issue of table retrievals is not specifically addressed by WIMA, because it requires a change in the information model. In WIMA, we do not define our own information model: we want to be able to use all possible information models, existing and future ones. But the support for the `noSuchInstance` error code can easily be added to a management gateway.

2.4.1.4 Poor efficiency of BER encoding

The SNMP protocol (v1, v2, and v3) uses BER [110] 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 both encoding and decoding. But the encoded data is verbose: the amount of administrative data (identifier and length) is large compared to the payload (content). This makes the network overhead unnecessarily large. It also increases the end-host latency, because more data takes more time to marshall and unmarshall. Mitra [147] and Neufeld and Vuong [154] describe in detail the performance issues with BER.

BER encoding was not designed with a view to reducing latency or network overhead: it was just simple to implement. This simplicity was the very reason why it was selected for SNMPv1. (Later, it was kept in the SNMPv2 and SNMPv3 protocols for backward compatibility). But the price to pay for this simplicity is the poor track record regarding network overhead and, to a lesser extent, latency.

The ISO has standardized three alternatives to BER. These are studied and compared by Mitra [147]. To summarize, Packed Encoding Rules (PER [111]) generate data that is approximately 30% more compact than

BER-encoded data, at the expense of a significant increase in the encoding time. Distinguished Encoding Rules (DER [110]) slightly improve the encoding time over BER, and have a low impact on network overhead compared to BER; but the gain offered by DER over BER remains marginal. Canonical Encoding Rules (CER [110]) are less demanding than PER in terms of encoding time, but encoded data is more verbose. CER is better than DER if the value to encode is so large that it does not entirely fit into the available memory. Note that LightWeight Encoding Rules (LWER [105]) have been investigated but not standardized by the ISO. They decrease latency by ensuring quick encoding and decoding. But LWER-encoded data can be much larger than BER-encoded data, which has a negative effect on network overhead.

As we can see, none of these encodings is clearly superior to BER. On a case-by-case basis, some are better than others; but none is always superior. As far as ISO encodings are concerned, there is always a trade-off between end-host latency and network overhead: you cannot win on both sides. If you gain on end-host latency, you lose on network overhead, and *vice versa*. A quick but simple encoding results in a smaller end-host latency but a larger network overhead, whereas a slow but smart encoding increases end-host latency while decreasing network overhead.

In WIMA, we solve this problem by allowing for any representation scheme and any encoding scheme in the communication model, instead of imposing BER. By compressing data, we significantly reduce network overhead while keeping latency almost unchanged (see end of Section 2.4.1.6).

2.4.1.5 Verbose OID naming

Another problem with SNMP-based management lies in the OID naming scheme used by the information model (that is, the naming conventions for MIB variables). When we take a close look at the OIDs that are transferred in SNMP messages, we observe a high degree of redundancy. For instance, all objects stored in MIB-II are prefixed with 1.3.6.1.2.1. If this prefix could be omitted, 20% to 50% of the OID name would typically be saved. More significantly, if we consider the objects stored in a table (these account for a large proportion of MIB variables), the prefixes of the OIDs are all identical up to the column number, and the index postfixes of all the entries of a single row are the same. In this case, more than 90% of the OID name is redundant. This naming scheme is really inefficient!

If we compare the OID naming scheme of SNMP with that of the Unix file system, we see that there are no concepts similar to *change directory* (`cd` command) and relative paths in SNMP: all OIDs are specified as absolute paths from the root `iso(1)`. The use of relative paths within a varbind list would make OIDs shorter, especially for tables, thereby decreasing the network overhead.

The second problem associated with OID naming is that each number composing the OID (that is, each sub-identifier) is BER-encoded separately—except the first two, which are coded on a single byte. This is inefficient. For instance, the 9-byte OID of `sysDescr` in MIB-II (1.3.6.1.2.1.1.1.0) is BER-encoded as 06:08:2B:06:01:02:01:01:01:00 (10 bytes). This leaves us with a compression ratio of about 1: the encoded data takes as much space as the initial data. By comparison, the compression ratios obtained with `gzip` on large files of textual data typically range from 3 to 5: the compressed data is 3 to 5 times smaller than the initial data.

For these two reasons, there is ample room for decreasing network overhead by simply changing the OID naming scheme of SNMP. Note that latency is also affected: if the OID naming scheme of SNMP were more efficient, the agent would not have to encode so many integers and the manager would not have to decode all of them.

In WIMA, we do not define a new naming scheme: this would require the specification of a new information model, and we want to be able to use any information model. Instead, we compress all the data, including OIDs, which reduces network overhead without affecting latency too much (see end of Section 2.4.1.6).

2.4.1.6 No compression of management data

Strangely enough, the designers of SNMPv1 worked hard to save every single byte to be sent over the network (e.g., see sparse tables), but they did not allow for the transparent compression of management data in transit. This unnecessarily increases network overhead and, as a result, network latency. One reason for this may be the *Myth of the Dumb Agent*: if the agent is "dumb" and short of resources, it does not have the CPU and memory resources to compress data dynamically.

As of SNMPv3, it is possible to compress management data by adding encryption envelopes to SNMP messages [201]. Although this feature was initially intended for encrypting data in transit, it also allows for data compression. By adding an encryption algorithm that compresses the message, the size of the messages that are transmitted over the network can be decreased substantially. Preliminary tests by Schönwälder showed that BER-encoded SNMPv3 PDUs can be compressed by 90% when using the DEFLATE format [62] and `get-bulk` operations[1]. By defining compression as an encryption algorithm, we can add compression to SNMPv3 without making any changes to the protocol. However, since there is no `noAuthPriv` security level in SNMPv3, one has to use authentication in order to take advantage of compression. This is a major limitation of this scheme, which was discussed to a great extent by the IRTF Network Management Research Group. We came to the conclusion that it made this scheme unusable in practice.

In WIMA, management data can be compressed, which significantly decreases network overhead. It increases end-host latency (the agent and the manager have to do more work), but significantly decreases network latency because there is less data to move about. Overall, compression is almost negligible with respect to latency.

2.4.1.7 Polling

In Section 2.4.3.1, we mentioned that in the SNMP management architecture, monitoring and data collection are based on polling, that is, on the pull model. This scheme is inefficient with respect to:

- *network overhead*: the manager keeps sending the same requests to the agents, which unnecessarily loads the network;
- *end-host latency at the manager*: the manager keeps putting together, encoding, and marshalling the same requests for sending them to the agents, which unnecessarily loads the manager;
- *end-host latency at the agent*: the agent keeps unmarshalling and decoding the same SNMP messages, which unnecessarily clutters the agent's CPU and memory resources; and
- *network latency*: the pull model requires a round trip whereas the push model requires a one-way transfer.

We will describe this problem and our proposed solution in great detail in Section 6.1.4.2.

In WIMA, this problem is solved by going from a pull to a push model, which involves changing the organizational and communication models of the management architecture. This is one of the most drastic changes that we propose.

2.4.1.8 Unreliable transport protocol

Another problem that we identified in SNMP-based management is the transport protocol used underneath the SNMP protocol. Why use UDP to send critical or fatal notifications to the manager (e.g., "interface XXX has gone down on backbone router YYY")? By using an unreliable transport protocol, we run the risk of losing important notifications for silly reasons such as buffer overflows in IP routers[2]. Why be exposed to the small-message-size problem described in Section 2.4.1.2, partially due to the use of UDP, when we could use TCP and large-size application-level messages?

---

1. Note that more tests should be performed to confirm this value in a general case.
2. Buffer overflows are a normal way to deal with bursts of traffic in the IP world. In case of buffer overflow, packets are silently dropped by the IP router (that is, the source of the packet is not notified that this packet was dropped).

When the SNMPv1 protocol was devised, the underlying transport protocol was not mandated. In the IP world, it could be either UDP or TCP. In practice, virtually all commercial offerings opted for UDP from day one, sometimes on technical grounds, but generally only because everybody else did (snowball effect). The technical grounds for the superiority of UDP over TCP were that TCP-based SNMP traffic would make a bad situation worse when a network is severely congested, because of TCP's three-way handshake and automated retransmissions: both increase network overhead. This rationale was defended for years by Rose, one of the designers of SNMP and a fervent opponent to using SNMP over TCP [176, pp. 20–22]. The relevance of this argument was destroyed by Wellens and Auerbach when they denounced the *Myth of the Collapsing Backbone* [242]. In short, most real-life networks are overprovisioned, and SNMP is of no use whatsoever for troubleshooting serious congestions of such networks. Instead, operators typically resort to tools such as `ping`, `traceroute`, etc. This counterargument is fully endorsed by the author, as it corroborates his experience in managing real-life networks.

We identified three good reasons for choosing TCP to transport management data. First, it significantly reduces data losses (we have no guarantee of delivery, though). Two facts are often overlooked by administrators: buffer overflows do occur in real life, and some management data is lost. The frequency of these losses is site specific and even network-link specific; it also varies considerably over time. Some administrators are unaware of the proportion of management data that never reaches the manager, including some critically important notifications. With TCP, notifications are less often lost *en route* due to buffer overflows in IP routers.

Second, TCP allows for very large application-level messages. This makes bulk transfers more efficient and diminishes network and end-host latency, as the number of messages exchanged over the network is significantly reduced (provided that the TCP window size does not change too often). TCP's window mechanism allows several chunks of data to be in transit in parallel. This removes the latency caused by additional round trips when a table row does not fit into a single UDP message, or when the requested *`max-repetitions`* parameter for a `get-bulk` request does not fit into a single UDP packet. Not only does latency decrease, but SNMP-table consistency also improves, as we shorten the time window during which the agent might update a table while the manager is recursively retrieving it.

A third reason for using a reliable transport protocol to transfer management data is to relieve the management application from managing all the timers and performing all the retries currently necessary to recover from network losses in the case of SNMP polling. In our view, this task does not belong in the application layer, but rather in the transport layer.

In Section 7.2.2, once we have elaborated on our solution, we will come back to this issue and present two other reasons for using TCP instead of UDP. For the sake of completeness, we should mention that the IRTF Network Management Research Group is currently working on an Internet-Draft proposing an SNMP-over-TCP transport mapping [184] as an alternative to the standard SNMP-over-UDP transport mapping [46]. This work is progressing and might eventually become an experimental RFC. But it is still confined to the research community, and there is no sign as yet that this SNMP-over-TCP mapping will be adopted by the SNMP industry.

In WIMA, we solve these problems by using TCP instead of UDP.

2.4.2 Missing features

Now that we have investigated the features that are not implemented optimally in SNMP-based management, let us identify the features that are missing altogether. We broke them down into two groups: those related to security and those related to the low level of semantics of SNMP.

2.4.2.1 Security

Two security-related features are missing in SNMP: the support for simple, transparent security and the possibility to go across firewalls easily.

No simple, transparent security

SNMPv1 and v2c support no security. Identification based on a *community string* is so simplistic that it cannot be counted as a security scheme. There is no authentication, and the community string is even transmitted in clear over the network!

The main novelty brought by SNMPv3 is security. SNMPv3 supports identification, authentication, encryption, integrity, access control, etc. This security requires significant configuration (keys management, VACM tables, etc.) and is anything but transparent to the end user. SNMPv3 does not support strong security: the keys can be stolen by an attacker [182]. In cryptography, SNMPv3's security scheme is considered weak because there are known ways to break it. For the purpose of NSM, we classify it as *medium security* because it offers sufficient security for most customers.

SNMP-based management gives us the choice between two levels of security:

- no security (SNMPv1 and SNMPv2c)
- medium security (SNMPv3)

In our view, there are actually four levels of security in NSM:

- no security
- simple security: transparent, *à la* Secure Sockets Layer (SSL [205])
- medium security: sophisticated key management, sufficient if you are not paranoid about security
- strong security: needed if you are paranoid about security (e.g., banks and the military)

If we compare our risk levels in NSM with those defined by Smith in a general context [197, pp. 60–61], our *simple security* roughly corresponds to his *low risk*, our *medium security* to his *medium risk*, and our *strong security* to his *high risk*. Smith defines yet another risk level, called *critical*, when the loss of life or a major disaster is at stake. We include this last category in *strong security*.

In NSM, people are mostly interested in our second and third levels of security: *simple security* and *medium security*. For most sites, these levels offer enough security, but no more than needed. In SNMP-based management, we have the first and third levels of security, but we miss the second and fourth levels. In Section 2.5, we will see that a management architecture should not rely on strong security; so it is not a problem if it is missing in SNMP-based management. But the absence of support for simple security causes a real problem. In NSM, we really need the support for simple, transparent security because this is what many enterprises want to use (especially SMEs). They do not want passwords to be transmitted in clear, as in SNMPv1 and v2c: they simply want transparent identification and authentication *à la* SSL. Some also want some kind of encryption, so that a casual attacker cannot easily access sensitive data (e.g., which customers pay for a gold service). But most of them do not really mind whether the confidentiality of management data is guaranteed, and have no problem if a smart attacker manages to eavesdrop management data in transit over a WAN link.

In WIMA, security is decoupled from our own communication model. We can use any level of security offered by HTTP: HTTP authentication, SSL, etc. We entirely rely on the security schemes supported by the W3C. For strong security, we also need data-link encryption hardware.

Firewalls are difficult to cross

Firewalls have not yet been taken into account by the SNMP designers. When the author discussed this issue with members of the IETF working groups in the past, he always got the same answer: "Why would you want to go across a firewall with SNMP?". In Section 6.2.4, we will present several scenarios justifying the need for management data to go across one or several firewall(s).

In WIMA, we addressed this problem by taking firewalls into consideration from the outset and changing the communication model accordingly.

2.4.2.2 Information model: low level of semantics

The need for high-level semantics in management will be described in great detail in Section 3.2. Let us briefly summarize it here. 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. Because of these limitations, writing smart management applications is difficult, so people often write low-level applications, very close to instrumentation. This partly explains why management applications are often limited to little more than monitoring in the IP world.

No high-level MIBs

In Section 2.2 (“MIBs offer instrumentation APIs”), we already pointed out the absence of high-level MIBs in SNMP-based management. This is a major problem for information modelers and management-application designers.

Too few SNMP protocol primitives

All three SNMP protocols (v1, v2, and v3) support basically three protocol primitives, if we group them by similar semantics: `get`, `set`, and `notify`. This is extremely limited and is typical of a data-oriented information model. In object-oriented software engineering, this design problem is known as the *Blob* antipattern [34] or the *God Class* problem [173]. It is a direct consequence of the *Myth of the Dumb Agent*.

SNMP information model: not object oriented

Object-oriented models are widely used in industry today, and it is a shame for an information modeler to be forced by SNMP to go back to old data-oriented modeling. Strangely enough, object-oriented modeling was already in the air in the late 1980s, but it was not adopted by the designers of SNMPv1. The absence of an object-oriented information model in SNMP is generally regarded as one of the main limitations of SNMP. When the DMTF endeavored to define a new management architecture in the late 1990s, it came as no surprise that its first delivery was a new object-oriented information model: the Common Information Model (CIM). We will present CIM in Section 5.5.1.

Programming by side effect

If we combine the limited number of SNMP protocol primitives with the data-oriented nature of the SNMP information model, we see that SNMP information modelers have a lot of difficulty implementing the equivalent of *behavior* (*methods*) in object-oriented software engineering. This problem has led to an ugly workaround known as *programming by side effect*.

To trigger an action on a remote agent, the manager sets an integer MIB variable (let us call it `doIt`) to a certain value. Different values trigger different actions: if you set `doIt` to 39, you reboot the machine; if you set `doIt` to 57, you reset an interface; etc. This very poor design often leads to “spaghetti code” in management applications: programmers do not use wrapper libraries offering a higher-level API, but hard-code the different values (39, 57, etc.) all over the code. This makes it impossible to maintain the management application in the long run. In software engineering, it is well known that a poorly designed API often leads to a poor design of the applications using it. It is unfortunate that SNMP does not provide management-application designers with a higher level of semantics.

In WIMA, we will propose several ways of alleviating or correcting this problem. One of them uses CIM and the eXtensible Markup Language (XML).

2.4.3 Nontechnical problems in SNMP-based management

In this section, we study the last category of problems that we identified in SNMP-based management: nontechnical problems. These problems are related to SNMP-based management platforms, to the need for domain-specific expertise, and to SNMP's slow evolution pace.

2.4.3.1 Problems with management platforms

The problems related to SNMP-based management platforms are not technical in nature: they are due to the way the management-platform market evolved over time. These problems are not inherent in the SNMP management architecture or the SNMP protocol. But as they have been going on for a decade, there is little hope of seeing them disappear from the SNMP market.

In 1998, we identified a number of problems by making an in-depth analysis of how SNMP-based management platforms typically work, and how these platforms evolved during the 1990s [132]. To summarize our methodology, we first studied how SNMP-based management platforms evolved from the specification of SNMPv1 to the advent of Web-based platforms. Then, we described the initial vision of an open market with generic tools, and explained how it evolved toward captive markets with vendor-specific tools. Finally, we analyzed the consequences of the arrival of Windows-based management platforms in a market previously dominated by Unix. The conclusions of this work are the following.

Customers' grievances

For customers, SNMP-based management platforms present four shortcomings. First, they are too expensive in terms of hardware and software. In particular, customers resent that in practice, a dedicated machine is needed to manage small networks. Second, they want to be able to store management data in whatever RDBMS they happen to own. They do not want to rely on preexisting peer-to-peer agreements between management-platform and database vendors (see Section 2.2, "Tightly coupled data repository"). Third, they want to be able to move the manager easily from one machine to another, e.g. from a Windows PC to a Linux PC, without having to pay for a new software package. Fourth, current management platforms offer insufficient integration of management (see Section 2.5).

In WIMA, the first, second, and fourth problems are solved. The third problem can be solved but not necessarily.

Vendors' grievances

Equipment vendors are primarily dissatisfied with the huge costs they have to bear to port their device-specific management GUIs (one for IP routers, one for level-3 switches, one for ATM switches, one for intelligent hubs, one for plain hubs, etc.) to all existing management platforms, that is:

- to all existing processors: Intel XX, Motorola YY, etc.
- to all existing operating systems: Linux XX, Windows YY, Solaris ZZ, etc.
- to all existing managers: HP OpenView, Cabletron Spectrum, etc.

This explosion of the number of different platforms to support (and test before each release of a new management GUI) leads to an exponential growth in the development and testing costs incurred by equipment vendors. Over time, add-on developers have learned to live with this heterogeneity, and wrapper code generally hides system-specific idiosyncrasies from the rest of the management application. This reduces the development costs, but definitely not the testing costs.

This problem is solved in WIMA by embedding Java management applets into network and systems equipment, and by relying on the portability of Java code.

Customers and vendors' common grievances

Finally, some grievances are shared by customers and equipment vendors alike. First, they all want to reduce the time-to-market of vendor-specific management GUIs. In SNMP-based management, there is often a time lag of several months between the time a new piece of hardware appears on the market and the time a customer can purchase an add-on for his/her favorite management platform. Ideally, customers and vendors would like any network device and any system to be manageable via a nice management GUI (not a mere MIB browser) as soon as it is launched on the market.

Second, in order to attract more customers, start-up companies want to have access to the major management platforms. So do their customers, because this enables them to drive the costs down by increasing competition. Usually, start-ups cannot afford the peer-to-peer agreements that give equipment vendors access to a management-platform API, and an entry in the catalog of the management-platform vendor. Neither can they afford the development costs for porting their management GUI to myriads of management platforms. Ideally, a start-up would like its customers to be able to download their add-ons into any management platform, without the start-up having to pay royalties to any management-platform vendor. Instead, in SNMP-based management, customers typically use a separate, dedicated management platform when they buy a piece of equipment from a start-up. This defeats the concept of integrated management. When the author was working in industry, this problem proved to be a major restriction when purchasing new equipment.

Third, both vendors and customers need a solution to the problem of MIB versioning. From time to time, equipment vendors release a new version of their proprietary MIB and management GUIs. As it is not possible to upgrade all the agents and the manager simultaneously, and as old pieces of equipment may not support the latest vendor-specific MIB, customers have to live with several versions of a MIB coexisting in the same network, either temporarily or permanently. Unfortunately, there is no MIB-discovery protocol in SNMP[1]. Thus it is not easy to maintain the manager's knowledge of an agent's MIB version synchronized with the actual version of this MIB in that agent. Some management platforms require that the manager be updated manually, others demand that it be updated externally via scripts; yet others do not allow several versions of a MIB to coexist in a manager.

In WIMA, all of these problems are solved by embedding management applets into network and systems equipment.

2.4.3.2 Domain-specific expertise

SNMP expertise is domain specific, hence rare and expensive. In particular, new programmers need specific training. This may seem absurd to a newcomer to SNMP-based management. If we compare the technical merits of SNMP (especially the SNMP protocol) with those of other existing technologies, they hardly compete because the design of SNMP is, by and large, outdated. Is SNMP another incarnation of the *Reinvent the Wheel* pattern [34]? It is not, for a very simple reason: in the IP world, it came first. When SNMPv1 was devised, none of the technologies that we use in WIMA existed. In the late 1980s, it therefore made sense to define a new protocol and a new management architecture: the existing ones (OSI management) were not appropriate. But now that the *corpus* of Internet technologies has grown enormously, and many of these technologies are routinely reused in different domains, it no longer makes sense to use a domain-specific technology for management, especially for the communication protocol. Transferring management data is not vastly different from transferring user data!

In WIMA, we solved this problem by using standard and ubiquitous Web technologies.

---

1. Agent capabilities (e.g., MIBs) can be documented in `sysORTable` [47]. In theory, this feature could be used to implement a MIB-discovery protocol. In practice, administrators cannot rely on it because it appeared only in 1996 with the second SNMPv2 generation (most deployed agents support SNMPv1). Moreover, among SNMPv2c- and SNMPv3-compliant agents, some do not maintain the `sysORTable` table up to date.

2.4.3.3 SNMP evolves too slowly

The last problem that we identified in SNMP-based management is its inability to evolve quickly enough to meet the market's demands in a timely manner. Despite the growing expectations and needs of the market (see Section 2.5), the IETF working groups in charge of SNMP have progressed too slowly in the 1990s. Two of the reasons for the initial success of SNMPv1 were that it was lightweight compared to OSI management, and it did not have to go through the four-year standardization cycles of the International Telecommunication Union—Telecommunication standardization sector (ITU-T). But experience has shown that SNMP evolves at an even slower pace. SNMPv1 was released in 1990. In 1994, SNMPv2p proved to be a complete failure [141, 166, 207, p. 334]. In 1996, SNMPv2c added very little to SNMPv1. In the end, it took eight years before the IETF delivered a substantial new release, SNMPv3, and another two years before major companies began supporting it[1]. By this time, the market had lost confidence in the ability of SNMP to meet its needs.

Despite all the good work that was done by the IETF SNMPv3 and DIStributed MANagement (DISMAN) Working Groups, it appears that many vendors and customers do not contemplate using SNMPv3 in the future. In November 1998, at the Networld+Interop trade show in Paris, France, the author asked many equipment vendors whether they already supported SNMPv3. Their answers were unanimous: "No, but who wants SNMPv3 anyway? You are the first to ask." In the next management cycle, it seems likely that SNMPv1 will survive unchanged as a simple solution for monitoring network equipment, while alternatives to SNMP will be used to perform advanced management tasks. We propose to use WIMA as an alternative to SNMP-based management.

2.5 We need a new solution to a new problem

In view of all the problems listed in Section 2.4, one could be tempted to criticize the designers of SNMPv1 for overlooking or poorly engineering so many things. However, this would be grossly unfair. First, we recalled in Section 2.3 the strengths and achievements of SNMP. SNMP has been a success story in network management. Today, most IP network devices in the world are managed with SNMP-based management platforms. Second, we should keep in mind that hindsight-based analysis is an easy art. It is simple to analyze retrospectively how things should have been designed in the first place. It is considerably more difficult to make the same analysis beforehand. Third, the requirements for managing IP networks and systems have changed considerably since the inception of SNMP, a decade ago. The problem that we are trying to solve today is different from the problem that was successfully solved in the late 1980s: no wonder SNMP is not appropriate for NSM today!

In this section, we substantiate this third argument by summing up the requirements that we identified for tomorrow's network and systems management. In the remainder of our dissertation, we will see how WIMA fulfills these requirements.

Scalability

The main requirement for tomorrow's NSM is that we *must* improve the scalability of the management architecture for the next management cycle. SNMP-based management is centralized, and there is no way it can cope forever with the continuous increase in the amount of management data to move about and process (see Section 2.4.1.1). There are many agents deployed worldwide that are not "dumb" at all and have plenty of memory and CPU cycles to use.

In WIMA, scalability was the first issue that we addressed. Our push model requires agents to do more than in SNMP-based management, but it does not impose on them the footprint of full-blown middleware like CORBA.

---

1. For instance, Cisco supports it as of IOS 12.1, officially released in March 2000 [57].

Cost

In its early days, SNMP-based management was reasonably inexpensive. The extra cost of adding an SNMP agent to a network device was small for equipment vendors, and the support for SNMP was a good selling argument, so vendors did not charge a lot for SNMP. The first management platforms on the market were perhaps minimalist and concentrated on monitoring, but they were inexpensive. Over time, the initial vision of open management in a commodity market evolved toward a mosaic of secure, niche markets where competition is seriously hampered by peer-to-peer agreements between vendors and costs remain artificially high [132]. With this new business model, the management-platform market has become very unfavorable to customers. The marketing concept of a *preferred business partner* basically means that sheer competition does not work and prices remain high.

In the next management cycle, the business model of the management-platform market should be changed to reduce the costs for both equipment vendors and customers, and to place start-up companies on a par with larger equipment vendors. Some customers want to pay less for the same service, that is, only integrate network and systems management in a single management platform. Other customers want to pay the same amount of money, or only a little bit more, for a better service: they want full-blown integrated management (see next section “Integrated management”). To meet these demands, vendors must find a way to reduce the amount of money that they charge per management task. To do so while preserving their margins, they must significantly cut the development, testing, and maintenance costs of their add-ons (vendor-specific management GUIs).

In WIMA, embedded management applets and component-based management platforms allow precisely for that.

Higher level of semantics

The third requirement for the next management cycle is to improve the user-friendliness of NSM in the IP world. When analyzed by a software engineer of the 2000s, the SNMP protocol and the SNMP management architecture seem outdated in many respects. Designers and programmers in industry are used to manipulating higher levels of semantics, and to working at a higher level of abstraction than what management-platform APIs typically offer them today in SNMP-based management (see Section 2.4.2.2). These people should be offered the same kind of user-friendly environments, whether they design and develop a distributed application for integrated management, aeronautics, or banking. Nothing inherent in management forces application designers and programmers to work at an instrumentation level: this habit in SNMP-based management is only due to its development history.

In WIMA, we either use CIM, the object-oriented information model devised by the DMTF during this Ph.D. work, or wrap low-level information models like SNMP’s with XML.

Security and firewalls

When SNMPv1 was devised, most enterprises did not care about network and systems security because the Internet was not what it is today. Security concerns have grown enormously during the 1990s, especially after attacks on well-known Web sites and email-based virus infections were largely advertised by the press. As a result, firewalls are now very common, and customers have become much more demanding in terms of security. They want management platforms that offer them secure management and management of security.

Another requirement imposed by most customers is that the management architecture for the next management cycle must not require strong security. Strong security can be implemented at different levels. Data-link encryption hardware is the most secure way of simultaneously guaranteeing access control, authentication, confidentiality, integrity, nonrepudiation, etc. It works by making the traffic incomprehensible to an eavesdropper. But apart from banks and the military, most enterprises cannot afford this type of hardware, which is very expensive, and therefore cannot be mandated in a general case.

Alternatively, strong security can be based on strong cryptography. Cryptographic protocols work with keys, some public, others private [197, 208]. These keys pose two problems. The first problem is technical. Although the complex mathematics of cryptography is now reasonably well understood, the more mundane engineering issue of key management in a geographically dispersed enterprise is still considered an unsolved problem. The main issues at stake are key certification, key infrastructure, and trust models [175]. The second problem is commercial. How can we have all NSM vendors agree to abide by the same interoperable security standards in management when they have been selling proprietary security solutions for years? If we consider how long it takes major credit-card organizations to convince the industry to adopt and widely deploy a single standard (Secure Electronic Transaction) for secure e-commerce, in a business that depends entirely on interoperability to live, we can easily guess how difficult it will be in the management industry that can live happily without security interoperability... In consequence, it would be very unwise to make strong security mandatory in NSM.

In the next management cycle, management applications should be able to communicate across firewalls and support no security, simple security, and medium security (as defined in Section 2.4.2.1, “No simple, transparent security”). They should not require strong security, but they could support it as an option.

In WIMA, we actually changed our communication model to facilitate the traversal of firewalls.

Integrated management

The final requirement for managing tomorrow’s IP networks and systems is to support integrated management. We propose a three-phase integration path for the next management cycle.

First, we should integrate systems management across all markets: network devices, Windows PCs, Linux PCs, Unix workstations, Web-enabled handheld devices, etc. Today, systems management is to a great extent proprietary, especially for Windows PCs. Now that network management has successfully shown the way, systems management ought to move to open management technologies and architectures. Today, no technical reasons can justify why customers should use separate platforms for managing different types of systems.

Second, network and systems management should be integrated. This integration would not only reduce the running costs of NOCs, it would also allow for a more accurate event correlation. When systems management is decoupled from network management, we leave it to a human to correlate network problems (e.g., the crash of a router) with systems problems (e.g., the unreachability of a server). Rule-based systems can automate this task provided that NSM is integrated.

Third, for several years, customers have been requesting the full-blown integration of network, systems, application, service, and policy management—that is, integrated management. In particular, they want management platforms to swiftly integrate service management (especially dynamic service deployment), policy management, and QoS management. The rationale is that by integrating all the management tasks, administrators will be able to better automate management, to routinely support what is currently exceptionally offered (e.g., QoS management, dynamic service provisioning, etc.), and, in the end, to migrate from a largely reactive management paradigm to a more proactive paradigm.

In WIMA, we address the first two levels: integration of systems management, and integration of network and systems management. We decided to leave the third level of integration for future work, although preliminary investigations suggest that WIMA can easily be extended to application, service, and policy management.

2.6 Summary

This concludes our problem statement. In Section 2.1, we defined the terminology used in this dissertation. In Section 2.2, we reviewed the main characteristics of SNMP-based management and described the concepts of “dumb” agent, polling, vendor-specific management GUIs, etc. We saw that in the IP world, monitoring is not notification driven. In Section 2.3, we summarized the main strengths of SNMP-based management. Even though this Ph.D. work is primarily about replacing SNMP with a new management architecture, better suited

to today’s requirements, it is important to keep in mind that SNMP-based network management has been a tremendous success in the 1990s. In Section 2.4, we identified the main problems in SNMP-based management. Some of them are due to the scalability of the management architecture and the efficiency of the communication protocol. Other problems result from the lack of support for certain important features. The third type of problems are nontechnical. They are related to management platforms, domain-specific expertise, and the capacity of SNMP to evolve in a timely manner. Finally, in Section 2.5, we explained why we need a new solution to a new problem, and summarized the main requirements for the next management cycle.

Chapter 3

OVERVIEW OF THE SOLUTION SPACE[1]

How could we manage IP networks and systems tomorrow?

Our goal in this chapter is to provide administrators with sound technical grounds to choose management paradigms and technologies, and to take an evolutionary rather than a revolutionary approach to NSM. To do so, we endeavored to classify all open[2] technologies into a limited set of paradigms, and to propose criteria to assess and weigh the relative merits of different paradigms and technologies. One contribution of our work is to show that there is no win-win solution: different technologies are good at managing different networks and distributed systems. Evolving networks and evolving user requirements call for evolving NSM solutions. The aim of vendors is to sell revolutionary solutions, because they bring in more revenue in the short term. The purpose of administrators is to find an evolutionary path in the midst of these revolutionary approaches, to save money in the short and long run.

In this chapter, we introduce two ways of categorizing NSM paradigms; we call them the *simple taxonomy*[3] and the *enhanced taxonomy*. In Section 3.1, we present our simple taxonomy, based on a single criterion: the organizational model. In this taxonomy, all paradigms are grouped into four broad types: centralized, weakly distributed hierarchical, strongly distributed hierarchical, and strongly distributed cooperative paradigms. We then expose the strengths and weaknesses of this simple taxonomy, and explain why we need to enhance it. In Section 3.2.1, we draw a parallel between the ways in which enterprises and networks are organized; we delineate a common trend, and identify the delegation granularity as a criterion for our enhanced taxonomy. We then introduce the concepts of microtask and macrotask. In Section 3.2.2, we study the three other criteria retained for our enhanced taxonomy: the semantic richness of the information model, the degree of automation of management, and the degree of specification of a task. This leads us to our enhanced taxonomy, depicted in Section 3.2.3.

---

1. Early versions of the material presented in this chapter were published in a journal article [133] and a workshop paper [131].
2. This chapter does not cover proprietary solutions such as Microsoft's DCOM.
3. In organization theory, people generally refer to *typologies* rather than *taxonomies* when they mean *classifications by types* [95]. So did we, in early versions of this work [130, 131]. Since then, it was pointed out to us that usage has opted for *taxonomies* in computer science at large and network management in particular, so we now use this word. In the meaning of interest to us, both words are synonymous.

3.1 Simple Taxonomy of Network and Systems Management Paradigms

With the terminology defined in Section 2.1 in mind, let us now introduce our simple taxonomy of NSM paradigms. When we built it, we tried to meet seven objectives:

- provide an intuitive categorization of NSM paradigms;
- identify a small number of types;
- clearly separate centralized from distributed paradigms;
- highlight the differences between traditional and new paradigms;
- distinguish paradigms relying on vertical and horizontal delegation;
- enable administrators and NSM application designers to find out easily what paradigm is implemented by a given technology; and
- help classify quickly a new NSM technology.

To keep this taxonomy simple, and thereby meet the first two objectives, we decided to base it on a single criterion: the organizational model. This is the approach taken by most authors [97, 124]. To meet the third objective, we began with two types: centralized paradigms and distributed paradigms. *Centralized paradigms* concentrate all the management-application processing in a single node, the manager, and reduce all the agents to the role of "dumb" data collectors[1] (see *The Myth of The Dumb Agent* [242]). *Distributed paradigms*, conversely, spread the management application across several machines.

To meet the fourth objective, we further divided the *distributed paradigms* type. By studying the different technologies that implement distributed management, we discovered that regardless of their idiosyncrasies, they could all be classified into two categories, according to the role played by the agents in the management application. We called them *weakly* and *strongly* distributed technologies; they implement weakly and strongly distributed paradigms.

Weakly distributed paradigms are characterized by the fact that the management-application processing is concentrated in only a few nodes. Usually, the network is split into different management domains, with one manager per domain. In this scenario, all the agents remain limited to the role of "dumb" data collectors. Another example is to keep a single manager but to make a few agents smarter than the others. In both cases, we have one or two orders of magnitude between the number of smart machines and the number of "dumb" machines. Only a small proportion of the machines are involved in the management-application processing.

Strongly distributed paradigms decentralize management processing down to every agent. Management tasks are no longer confined to managers: all agents and managers are involved. Many strongly distributed technologies have been suggested in the recent past. As we will explain in Section 3.1.6, we found it natural to group them into three sets of paradigms: mobile code, distributed objects, and intelligent agents. The first two implement vertical delegation; we call them the *strongly distributed hierarchical paradigms*. The third implements horizontal delegation; we call this family the *strongly distributed cooperative paradigms*. This distinction allows us to meet our fifth objective.

Thus our simple taxonomy consists of four types:

- centralized paradigms
- weakly distributed hierarchical paradigms
- strongly distributed hierarchical paradigms
- strongly distributed cooperative paradigms

1. Actually, agents are not always "dumb". They can send unsolicited notifications to managers, and some can even process an atomic SNMP `set` of several MIB variables, which is not a trivial task. But the bulk of their activity is to collect usage data and be polled by managers, whereas the manager does all the real management work.

We refer to the first two as the *traditional paradigms*, and to the last two as the *new paradigms*. The strong distribution of the management application is a characteristic of new paradigms. The fourth type is also called *cooperative paradigms* for short, because the cooperative paradigms that we consider in NSM are always strongly distributed[1].

Our first five objectives have now been met. Let us delve into the details of these management paradigms and review the main technologies that implement them. We assume that the reader is familiar with the traditional management paradigms and protocols, that is, the different variants of SNMP, OSI management, Telecommunications Management Network (TMN), and Remote MONitoring (RMON)[2]. New paradigms and new protocols will be presented in more detail. At the end of this section, we will summarize our simple taxonomy in a synthetic diagram that will allow us to meet our sixth and seventh objectives.

3.1.1 Centralized Paradigms

Centralized paradigms are characterized by a single manager concentrating all the management-application processing, and a collection of agents limited to the role of "dumb" data collectors. The two typical examples are SNMP-based management and HTTP-based management (based on the HyperText Transfer Protocol).

3.1.1.1 SNMP-based management

To date, in the IP world, most real-life networks and systems are managed with centralized platforms based on SNMPv1 [42]. The success of this management architecture has been phenomenal. Within a few years, the networking industry, which was entirely dominated by proprietary equipment and management during the 1980s, turned to open systems and open management. Undeniably, the management of IP networks has been one of the greatest successes of open systems.

Nonetheless, three independent evolutions soon exposed a major weakness in the SNMPv1 management architecture, and more generally in the centralized paradigm: scalability. First, the IP world has been expanding for years at a very fast pace. Once limited to Unix machines, the TCP/IP stack became available on most network devices in the early 1990s, and on most PCs in the mid-1990s. Today, it is virtually ubiquitous. Second, the size of networks has grown dramatically. The number of PCs installed worldwide increased by several orders of magnitude during the past decade. The proportion of networked machines is now close to 100%, whereas many PCs were standalone machines when the SNMPv1 specification was released in 1990. Third, the size of SNMP MIBs increased, too. In LANs or WANs, IP routers and hubs had just a few FDDI (Fiber Distributed Data Interface), Ethernet, and Token Ring ports to manage a few years ago. Now, IP switches, ATM switches, and intelligent hubs have many more entities to manage (ports, cross connections, virtual LANs, etc.), and require much more data to be brought back to the manager.

The very success of the SNMPv1 architecture was the cause of its decline. It proved to be good at managing relatively small networks, but could not scale to large networks (e.g. geographically dispersed enterprises), and could not cope with ever more management data. A new paradigm was needed to address scalability. The telecommunications world had already shown how to solve this problem: by distributing the load across a hierarchy of managers (see next section). But strangely enough, the distribution of management was not a priority at the IETF until the late 1990s. Since SNMPv1, four management architectures have been released: SNMPv2p, SNMPv2u, SNMPv2c, and SNMPv3. The first three only support centralized management. SNMPv2p has been rendered obsolete by the IETF in 1996 [166]. SNMPv2u had little success and "saw no significant commercial offering" [166, p. 14]; 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 was on security [206], not scalability. We saw in Section 2.4.3.3 that major vendors only began supporting it in 2000. Its use is thus

1. This is not necessarily true in other fields; e.g., in DAI, some forms of cooperation can rely on a central entity.
2. These are presented by many authors [97, 124, 176, 195, 204, 207]. Perkins [166] wrote a good summary of the variants of SNMPv2.

expected to remain marginal in production environments in the foreseeable future. Note that the MIBs adding the support for one kind of distribution in SNMPv3 were issued only in 1999, 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.

3.1.2. HTTP-based management

Since the mid-1990s, with the Web becoming ubiquitous, inexpensive, and so easy to use, many people have argued that Web technologies were the way to go for NSM in the IP world [100]. New vendors, most notably start-ups, saw an opportunity to enter the lucrative market of management platforms. This led to a large family of approaches called *Web-based management*, which supports different management paradigms. In this section, we describe the approaches implementing a centralized paradigm. In Section 3.1.6, we will present those implementing strongly distributed hierarchical paradigms. Note that Web-based management will be revisited in great detail in Chapter 5, so we remain concise here.

HTTP-based management consists in using HTTP (either HTTP/1.0 [21] or HTTP/1.1 [74]) instead of one of the three SNMP protocols to transfer management data between agents and managers. For this to work, all agents must have an HTTP server embedded.

The simplest form of HTTP-based management relies on simple Web pages written in HyperText Markup Language (HTML) [137, 150]. The manager retrieves HTML pages from the agent and displays them in a Web browser, without processing them any further. Agents can send two types of documents: static and dynamic HTML pages. Static pages do not change over time and are stored in the agent (e.g., in Erasable Programmable Read-Only Memory, or EPROM for short). A typical example is a Web page for configuration management. Dynamic pages are generated on-the-fly by the agent in reply to a request received from the manager. They reflect the state of the agent at a certain time. A typical example is a Web page for performance management.

A second form of HTTP-based management consists in running an applet in a Web browser, or a Java [90] application, on the manager side, and using HTTP to communicate between the manager and the agent [18, 134, 242]. Management data can be pushed by the agent or pulled by the manager. Within HTTP, the data can be encoded in XML, HTML, strings, etc.

An alternative to this consists in using SNMP instead of HTTP to communicate between the manager (Java applet or application) and the agent. Bruins calls it *Java-based SNMP* [35]. Clearly, this case cannot be classified as HTTP-based management. This approach changes very little compared to standard SNMP-based management platforms: the technology used to build the GUI is different, but the same communication protocol is used underneath. The agent cannot tell whether it is communicating with a traditional SNMP-based or a Java-based management platform.

Compared with SNMP-based management, there is no change whatsoever in all these scenarios with respect to the management paradigm. We only change the communication protocol between the manager and the agent.

3.1.3 Weakly Distributed Hierarchical Paradigms

Weakly distributed hierarchical paradigms spread the management application across several machines. The telecommunications world has followed this management paradigm for years with TMN. In the IP world, we saw one failed attempt with SNMPv2p and one successful, but partial, attempt with RMON.

3.1.4. In the telecommunications world

Unlike the SNMP management architecture, which proved to be successful in many sectors of activity, the OSI management architecture [49, 50, 250] encountered very little success in data networks. But in 1992, the ITU-T adopted it as the basis for its TMN model [108, 190] and for the specification of some of the TMN interfaces that mandate the use of CMIP [52] and CMIS [51]. Since then, OSI management has flourished in the niche market of telecommunications networks, where it is used to manage both networks and systems.

TMN/OSI is based on a weakly distributed management paradigm that distributes management along the hierarchical tree of the managers, each in charge of a management domain. If the contact is lost between a mid-level manager and the top-level manager, independent corrective actions can be undertaken by the mid-level manager. If the contact is lost between a mid-level manager and an agent, the agent is left on its own.

One of the management services offered by CMIS is `M_ACTION`. It allows for the delegation of very simple tasks from a manager to an agent. In practice, this service is rarely used. But it is conceptually rich: any agent can execute a static, pre-defined task when requested by the manager. This gives us a flavor of strongly distributed management.

3.1.5. In the IP world

To date, the IP world is still waiting for a viable solution for distributing management across a hierarchy of managers. The first attempt was made in April 1993, when a new management architecture now called SNMPv2p was issued [83]. It relied on a new protocol and three new MIBs. Distributed management was supposedly made possible by a new protocol primitive, `inform`, and the Manager-to-Manager (M2M) MIB [43]. We call it the *SNMPv2p+M2M management architecture*. This architecture was primarily targeted at geographically dispersed enterprises. But the SNMPv2p security model (based on the concept of *party* [142]) and the M2M MIB were flawed and proved to be “unworkable in deployment” [141]. In 1996, SNMPv2p was superseded by SNMPv2c and SNMPv2u, both of which only support a centralized management paradigm.

Nonetheless, the IETF managed to successfully define a weakly distributed management paradigm by making some agents smarter than others. RMON probes were the first and simplest form of delegation added to the SNMPv1 architecture. They supported the RMON1 MIB, issued in 1991 [234] and updated in 1995 [235]. RMON2 was released in 1997 [236] and became widely supported by intelligent hubs and switching hubs. By gathering usage statistics in RMON-capable equipment, administrators can delegate simple network-management tasks to these specialized network devices, thereby relieving the manager of the burden of the corresponding processing and decreasing the amount of management data to move about. Tasks achieved by RMON are static, in the sense that only the gauges and traps hard-wired in the RMON MIB are available, together with all kinds of combinations thereof (via the *filter* mechanism). If contact is lost between the RMON-capable agent and the manager, statistics are still gathered, but no independent corrective action can be undertaken by the agent. RMON is well suited to manage fairly active LANs, and is widely used today in the IP world.

3.1.6 Strongly Distributed Hierarchical Paradigms

Weakly distributed hierarchical paradigms address the main shortcoming of centralized models, scalability, but they also exhibit a number of limitations. First, they lack robustness. If the contact is lost between the agent and the manager (e.g., due to a network link going down), the agent has no means to take corrective action in case of emergency. Second, they lack flexibility. Once a task has been defined in an agent (via RMON, or CMIP/CMIS with `M_ACTION`), there is no way to modify it dynamically: it remains static. Third, they can be expensive in large networks. By concentrating most of the management-application processing in managers, they require powerful (hence expensive) management stations.

To address this, a new breed of technologies emerged, based on strongly distributed hierarchical paradigms. The full potential of large-scale distribution over all managers and agents was first demonstrated in NSM by Goldszmidt with his Management by Delegation (MbD) architecture [88], which set a milestone in this research field. The novelty of his work stems on the simple, yet insightful, idea that with the constant increase in the processing power of every computer system and network device, NSM no longer ought to be limited to a small set of powerful managers: all agents could get involved and become active in the management application. For the first time with MbD, network devices were suddenly promoted from "dumb" data collectors to the rank of full-fledged managing entities.

MbD triggered a lot of research in strongly distributed NSM. The impact of the novel concepts it brought to this community was taken advantage of by many promising technologies that emerged, at about the same time, in other research communities. Most of these technologies came from software engineering, especially from the object-oriented and the distributed-application communities. Let us now present the paradigms underlying these technologies. They are grouped in two broad types: mobile code and distributed objects.

3.1.6.1 Mobile code

Mobile-code paradigms encompass a vast collection of very different technologies, all sharing a single idea: to provide flexibility, one can dynamically transfer programs into agents and have these programs executed by the agent. The program transfer and the program execution can be triggered by the agent itself, or by an entity external to the agent such as a manager or another agent.

Remote evaluation, code on demand, and mobile agents

Fuggetta *et al.* [82] made a detailed review of mobile code, where they clearly define the boundaries between technologies, paradigms (what they call *design paradigms*), and applications. As far as mobile-code technologies are concerned, they define *strong mobility* as the ability of a Mobile-Code System (MCS) to allow an execution unit (e.g., a Unix process or a thread) to move both its code and its execution state to a different host. The execution is suspended, transferred to the destination host, and resumed there. *Weak mobility*, on the other hand, is the ability of an MCS to allow an execution unit on a host to dynamically bind code coming from another host. The code is mobile, but the execution state is not preserved automatically by the MCS. (It is still possible to program this preservation explicitly, of course.)

By analyzing all existing MCSs, Fuggetta *et al.* identified three different types of mobile code paradigms[1]:

- *Remote EValuation* (REV [210]): When a client invokes a service on a server, it not only sends the name of the service and the input parameters, but also sends along the code. So the client owns the code needed to perform the service, while the server owns the resources. This is a form of *push*.
- *Code On Demand* (COD): A client, when it has to perform a given task, contacts a code server, downloads the code needed from that server, links it in dynamically (dynamic code binding), and executes it. Thus the client owns the resources and the server owns the code. This is a form of *pull*.
- *Mobile Agent*: It is an execution unit able to autonomously migrate to another host and resume execution seamlessly. So the client owns the code, while the servers own the resources and provide an environment to execute the code sent either by the client or another server.

A number of technologies can be used to implement these three paradigms. Some of them are just languages; others are complete systems that possibly include a virtual machine, a secure execution environment, etc. Agent Tcl, Ara, Emerald, Sumatra, Telescript, and Tycoon are examples of strong MCSs, whereas Aglets, Facile, Java (or rather, to be precise, a Web browser integrating a JVM—Java Virtual Machine—and supporting Java applets), M0, Mole, Obliq, and Tacoma are examples of weak MCSs (see references in [82]). Some technologies can be used to implement several paradigms [82].

1. These should be compared with the client-server paradigm, where the client invokes a service on a server, while the server owns both the code and the resources (that is, it provides the environment to execute the code).

Management by delegation and variants

Mobile-code paradigms were first used in NSM by Goldszmidt and Yemini, in 1991, when they devised Manager-Agent Delegation [251]. This management architecture was later enhanced and renamed Management by Delegation (MbD); it was fully specified in 1995 [88]. MbD is a mixture of the REV paradigm (to send delegated agents to elastic servers) and the client-server paradigm (to remotely control the scheduling and execution of delegated agents).

Burns and Quinn [39] were among the first to describe a prototype of a mobile agent used in NSM. Since then, MCSs have encountered a growing success in NSM [15, 24, 25, 33, 181]. Baldi and Picco [16] studied the network traffic generated by MCSs implementing REV, COD, and mobile-agent paradigms, and made a quantitative theoretical evaluation of the effectiveness and suitability of mobile-code paradigms in network management. The ISO integrated mobile-code concepts in its OSI management architecture by specifying a new management function: the Command Sequencer [113]. In 1999, the IETF DISMAN Working Group defined a series of MIBs allowing managers to distribute tasks to agents with SNMPv3. The most relevant to us is the Script MIB [126, 185], which allows a manager to delegate management tasks to an agent in the form of scripts. We call it the *SNMPv3 + Script management architecture*.

Active networks

One area where mobile code paradigms have recently encountered a large success is known as *active networks*. There are two approaches to active networks. The evolutionary approach, called the *programmable switch* or the *active node*, provides a mechanism for injecting programs into network nodes to dynamically program them [4, 252]. These programs may perform customized computations on the packets flowing through them, and possibly alter the payload of these packets (e.g., compress and decompress data at the edges of the network). Clearly, this breaks the principle that transport networks should opaquely carry user data. The revolutionary approach, also known as the *capsule* or *smart packet*, considers packets as miniature programs that are encapsulated in transmission frames and executed at each node along their path [224].

The concept of active networks was first proposed in 1995 by Tennenhouse and Wetherall [224]. It was first applied to network management by Yemini and da Silva in 1996 [252]. Research is now extremely active in this field [36, 118, 159, 170, 172, 188, 198]. In NSM, active networks are of great interest because they bring in flexibility and robustness. Network monitoring and event filtering [72, 223] are especially good candidates, as monitoring programs can easily be dispatched through the network. These programs are high-level filters that watch and instrument packet streams in real time. They maintain counters and report results back to the manager. Another example is active congestion control [72]. The main problems associated with active networks are security, performance, and interoperability [159]. A lot of work is under way to solve them, including coupling active software with active hardware in the area of Field-Programmable Gate Arrays (FPGAs) [99].

3.1.6.2 Distributed objects

Parallel to mobile code, a second type of strongly distributed hierarchical paradigms has emerged, based on distributed object technologies. We describe the four main approaches in this section: the Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), Web-Based Enterprise Management (WBEM), and the Open Distributed Management Architecture (ODMA).

CORBA

Faced with the issue of interoperability in the object-oriented world, the Object Management Group (OMG) standardized the Object Management Architecture, now commonly referred to by its main component: CORBA [191]. CORBA 2.0 [158] was released in 1995. Unlike its predecessors, this release proved very successful in the software-engineering community, particularly for large corporations with huge investments

in legacy systems. Because OSI is object-oriented and SNMP managed objects can be mapped onto objects, it took little time for NSM researchers to begin working on the integration of CORBA with existing management architectures. Pavlou was among the first when he proposed to use CORBA as the base technology for TMN [161, 163].

The Joint Inter-Domain Management (JIDM) group, jointly sponsored by the Open Group and the Network Management Forum (NMF)[1], was created to provide tools that enable management systems based on CMIP, SNMP, and CORBA to work together. The SNMP/CMIP interoperability was previously addressed by the ISO-Internet Management Coexistence (IIMC) group of the NMF, which specified the translation between the SNMP and CMIP/CMIS services, protocols, and information. Both CMIP/CORBA and SNMP/CORBA [138] interworking were solved by JIDM, who addressed specification translation and interaction translation. Algorithms were defined for the mapping between GDMO/ASN.1 (Guidelines for the Definition of Managed Objects / Abstract Syntax Notation 1) and CORBA IDL (Interface Definition Language) [139], and between SNMP SMI and CORBA IDL [140]. The JIDM mappings allow CORBA programmers to write OSI or SNMP managers and agents, without any knowledge of GDMO, ASN.1, or CMIP. Inversely, these mappings also allow GDMO, CMIS or SNMP programmers to access IDL-based resources, services or applications, without knowing IDL.

CORBA has also been integrated with Web technologies. One example from the industry is IONA's OrbixWeb, a Java Object Request Broker (ORB) coded as an applet. Once loaded into a Web browser, it runs as a CORBA server communicating via the Internet Inter-ORB Protocol (IIOP). Another example is CorbaWeb [145], from academia.

CORBA has been well accepted in the telecommunications world, where it is becoming a *de facto* standard, a rarity in this industry traditionally based on *de jure* standards. One of the achievements of the Telecommunications Information Networking Architecture (TINA [19, 20]) has been to add a DPE to TMN, and CORBA proved to be a natural choice for the DPE [164, 165]. Several major equipment vendors are now turning to CORBA to manage their telephone switches and networks.

Java RMI

Java RMI makes it possible to program a management application as a distributed object-oriented application. Everything is an object, and all objects can interact, even if they are distant (that is, if they are running on different machines). When Java RMI is combined with Object Serialization, which allows the state of an object to be transferred from host to host, management-application designers have a powerful technology at their disposal whereby they can mix mobile-code and distributed-objects paradigms.

Distributed Java objects can be mapped directly onto SNMP or OSI managed objects. In this case, they are low-level. They can also be high-level; e.g., the set up of a Virtual Path (VP) across multiple ATM switches can be defined by a single method call on a remote object living in the source switch. This high-level object will take care of all subsequent method invocations on all switches along the path to the destination. We will discuss this important concept in Section 3.2.

To date, Sun Microsystems and the Java Community have released three specifications leveraging Java RMI in NSM: the Java Management Application Programming Interface (JMAPI [221]), the Java Management eXtensions (JMX [218]), and the Federated Management Architecture (FMA [220]). These three aproaches will be described in Section 5.6. JMAPI is now obsolete, but it was a successful proof of concept. JMX has concentrated so far on the agent side and FMA on the manager side of the management application. These two frameworks are expected to converge in the future.

1. Since then, the NMF has become the TeleManagement Forum (TMF).

JMAPI, JMX, and FMA have shown that distributed Java-based management allows for a very powerful way of building a strongly distributed management application. But it mandates that all managers and agents support Java, or that all agents be accessed via a Java-capable management gateway, which is a strong requirement. FMA also relies on Jini, which is even more demanding.

WBEM

In the IP and telecommunications worlds, open standards have virtually wiped out proprietary NSM solutions, due to the large success encountered by the SNMP and OSI/TMN management architectures and protocols in these markets. But in the rest of the industry, proprietary management platforms are still the rule and open platforms are the exception. A few years ago, the DMTF issued the Desktop Management Interface (DMI [63]) specification and tried to promote open management for desktops. But this effort encountered little success [38, p. 2]. To date, most networked desktops are either unmanaged or managed with proprietary solutions.

The situation could soon change, however. Enterprises are now bearing the costs of two parallel management platforms: one to manage their IP-based network equipment, and another to manage their desktop PCs. These companies, who did not attach much importance to open management a few years ago, are now paying a high price for this lack of interoperability; and they are pushing the industry to integrate the management of all kinds of network and systems equipment: PCs, routers, printers, switches, etc.

To address this need, a new management architecture called Web-Based Enterprise Management (WBEM) is currently defined by the DMTF [69]. WBEM will be presented in detail in Section 5.5.1. WBEM has evolved quite a lot since its early days in 1996. So far, the main outcome of this effort has been the specification of a new information model: the Common Information Model (CIM [38]). Its main strength is that it is object-oriented, unlike the SNMP information model. Its main drawback is its terminology, which departs radically from the SNMP and OSI-management terminologies and mixes up database and object-oriented concepts such as schema and model. WBEM's communication model relies on two standard technologies: HTTP for the communication protocol, and XML for representing management data in HTTP messages. So, unlike CORBA and Java RMI, the communication between distant objects does not rely on a distributed object-oriented middleware, but on a serialization via HTTP. WBEM is backed by most vendors in the NSM industry; it is likely to emerge as one of the main management architectures of the decade.

ODMA

The purpose of ODMA [106] is to extend the OSI management architecture (thus also the TMN architecture) with the Reference Model of the ISO Open Distributed Processing (RM-ODP) architecture, which provides for the specification of large-scale, heterogeneous distributed systems. This joint effort of the ISO and the ITU-T has led to a specialized reference model for the management of distributed resources, systems, and applications. It is based on an object-oriented distributed management architecture composed of computational objects. These objects offer several interfaces, some of which are for the purpose of management.

In ODMA, there are no longer managers and agents with fixed roles, like in the OSI management architecture. Instead, computational objects may offer some interfaces to manage other computational objects (manager role), and other interfaces to be managed (agent role). Moreover, by adopting the computational viewpoint of ODP, ODMA also renders the location of computational objects transparent to the management application. As far as the management application is concerned, computational objects may live anywhere, not necessarily inside a specific agent or manager. Consequently, agents may execute advanced management tasks, like managers. In short, the ISO and the ITU-T have gone from a weakly distributed management paradigm, with the OSI management architecture, to a strongly distributed management paradigm, with ODMA.

3.1.7 Strongly Distributed Cooperative Paradigms

Unlike centralized and hierarchical paradigms, cooperative paradigms are *goal-oriented*. What does this mean? For example, in REV-based mobile code technologies, agents receive programs from a manager and execute them without knowing what goal is pursued by the manager. Managers send agents the "how", with a step-by-step *modus operandi* (coded in the program), and keep the "why" for themselves. Agents execute the program without knowing what it is about, they are "dumb". Conversely, with intelligent agents, managers just send the "why", and expect agents to know how to devise the "how". In this sense, agents used in cooperative paradigms are "intelligent". Obviously, there is a price to pay for this: cooperative technologies are much more complex to implement than centralized or hierarchical technologies. They also consume considerably more resources (processor, memory, and network bandwidth).

Cooperative technologies were only recently considered by the distributed NSM community. Until recently, most NSM authors simply ignored them [88, 97, 124, 146, 195, 207]. They originate from DAI, and more specifically from Multi-Agent Systems (MASs), where people are modeling complex systems with large groups of intelligent agents. This research field is fairly recent, so its terminology is still vague. Specifically, there is no consensus on the definition of an intelligent agent. Many authors have strong and different opinions about this (Franklin and Graesser [79] listed 11 definitions!), which does not help. In 1994, Wooldridge and Jennings took a new approach. Instead of imposing on others what an intelligent agent should or should not be, they defined a core of properties shared by all intelligent agents, and allowed any other property to be application specific. This approach has encountered a great deal of success, and contributed significantly to the dissemination of MASs outside the realm of DAI. For these authors, intelligent agents (or, to be precise, what they call *weak agents*) must exhibit four properties [248]:

- *autonomy*: An intelligent agent operates without direct human intervention, and has some kind of control over its actions and internal state.
- *social ability*: Intelligent agents cooperate with other intelligent agents (and possibly people) to achieve their goals, via some kind of agent-communication language.
- *reactivity*: An intelligent agent perceives its environment, and responds in a timely fashion to changes that occur in it.
- *proactiveness*: An intelligent agent is able to take the initiative to achieve its goals, as opposed to solely reacting to external events.

Proactiveness is a very discriminating property. While most intelligent-agent implementations are reactive, only a few of them qualify for proactiveness, particularly outside the AI community. We believe that this is the main difference between mobile agents from the software engineering community, and intelligent agents from the DAI community.

For Wooldridge and Jennings, optional properties of weak agents include mobility, veracity (intelligent agents do not knowingly communicate false information), and rationality (intelligent agents are not chaotic, they act so as to achieve their goals). In addition, they define *strong agents* as weak agents modeled with human-like characters, e.g. by using Rao and Georgeff's Belief, Desire, Intention (BDI) model [171]. Strong agents are the type of intelligent agents generally used by the DAI community, whereas weak agents are the type often used by other research communities.

Two years later, Franklin and Graesser [79] compared the approaches taken by many authors and, as Wooldridge and Jennings, distinguished between mandatory and optional properties. For them, intelligent agents must be reactive, autonomous, goal-oriented (proactive, purposeful), and temporally continuous (an intelligent agent is a continuously running process). Optionally, they can also be communicative (that is, able to communicate, coordinate, and cooperate with other agents,), learn (they improve their skills as time goes by, storing information in knowledge bases), be mobile, and have a human-like character. In our view, the fact that intelligent agents should be continuously running processes is an important property. It also distinguishes intelligent agents from mobile agents.

Because we consider intelligent agents in the context of cooperative paradigms in distributed NSM, their ability to communicate, coordinate, and cooperate should be, in our view, a mandatory property. The ability to learn is often expected from intelligent agents in NSM; but like many authors, we do not consider this property to be mandatory. We therefore propose that in distributed NSM, intelligent agents should always be:

- goal-oriented (proactive)
- autonomous
- reactive
- cooperative (communicative, coordinating)
- temporally continuous

When intelligent agents are cooperative, they are exposed to heterogeneity problems, and therefore need standards for agent management, agent communication languages, etc. Two consortia are currently working on such standards: the Foundation for Intelligent Physical Agents (FIPA [76]) and the Agent Society [2]. Among all the agent communication languages that emerged in DAI [248], the Knowledge Query and Manipulation Language (KQML [75]) and the Agent Communication Language (ACL [77]) have encountered a certain success in the distributed NSM community.

More and more researchers are now trying to use intelligent agents to manage networks and systems [149, 168, 199, 253]. But we should remember that the limits between mobile agents (following a mobile-code paradigm) and intelligent agents (following a cooperative paradigm) are sometimes fuzzy. And when Knapik and Johnson [120] advocate the use of *OO agents* (object-oriented agents), to combine the advantages of both worlds, the classification becomes even trickier. In fact, OO agents implement two paradigms simultaneously, distributed objects and intelligent agents, and can even implement a third: mobile code.

3.1.8 Synthetic Diagram

Our simple taxonomy is summarized in the following synthetic diagram:

	centralized paradigms	hierarchical paradigms	cooperative paradigms
not distributed	SNMPv1, SNMPv2c, SNMPv2u, SNMPv3, WBEM, HTTP		
weakly distributed		SNMPv1 + RMON, SNMPv2p + M2M, OSI/TMN	
strongly distributed		SNMPv3 + Script, mobile code, Java RMI, distributed objects	intelligent agents

Table 1. Simple taxonomy of NSM paradigms

The chief contribution of this simple taxonomy is that it highlights some similarities between apparently very diverse approaches. Despite the fact that new technologies appear at a fast pace, network and systems administrators are no longer overwhelmed by the variety of approaches offered to them: they have a simple way to analyze them and group them, which reduces the scope of their investigation.

The main disadvantage of this simple taxonomy is that it is more of academic than industrial interest. By only considering the organizational model, it remains theoretical, and does not give many clues as to what paradigm or technology should be used in the context of a given enterprise. Administrators and designers of management applications need more pragmatic criteria. They have difficult software-engineering decisions to make at the

analysis and design levels. They need to think twice before investing in expensive technologies such as CORBA, or before embarking for uncharted territories inhabited by roaming intelligent agents! Administrators would rather base their choices on sound technical grounds. Ideally, they would like case-based studies or "cookbook recipes". The purpose of our enhanced taxonomy is precisely to fulfill this need.

3.2 Enhanced Taxonomy of Network and Systems Management Paradigms

The first criterion of our enhanced taxonomy, delegation granularity, is derived by comparing the organizational models used in NSM with organization structures considered in enterprise management. The three other criteria were identified by comparing the technologies introduced in the previous section.

3.2.1 A stroll through organization theory

The topology of an enterprise computer network tends to be modeled after its organization chart. The main reason for this is that the people accountable for the smooth operation of these networks and systems belong to this chart, and it makes their life a lot easier if different managers (people, this time) have a hold on different computers and network devices. In addition, such a network topology is often justified in terms of budget: different departments pay for their own equipment. Sometimes, it also makes technical sense, for instance when different departments are located on different floors or in separate buildings. In short, NSM is not orthogonal to enterprise management.

In this section, we show that the first criterion of our enhanced taxonomy was derived by comparing the organizational models in NSM with organization structures considered in enterprise management. To do so, we studied how delegation works in enterprises, how it maps onto organization structures, and how the two fundamental paradigms that we identified in NSM (delegation and cooperation) map into the enterprise world.

3.2.1.1 Organization Structures in Enterprise Management

Mullins [152] distinguishes eight ways of dividing work in an enterprise:

- by function (one department per function: whether in production, R&D, marketing, finance, or sales, all staff share a common expertise within a department);
- by product (autonomous units, all functions are present in each unit);
- by location (geographically dispersed companies, subsidiaries abroad);
- by nature of the work to be performed (e.g., by security clearance level);
- by common time scales (e.g., shift work vs. office-hours work);
- by common processes (e.g., share a production facility in the manufacturing industry);
- by the staff employed (e.g., surgeons, doctors, and nurses in a hospital); and
- by type of customer or people to be served (e.g., home vs. export sales).

For Mullins, delegation can take place at two levels: enterprise or individual. At the enterprise level, it is depicted in the organization chart (at least it is supposed to be!) and relies on federal or functional decentralization. *Federal decentralization* is defined as "the establishment of autonomous units operating in their own market with self-control and with the main responsibility of contributing profit to the parent body" [152, p. 276]. As for *functional decentralization*, it is "based on individual processes or products" [152, p. 276]. At the individual level, delegation is "the process of entrusting authority and responsibility to others" [152, p. 276] for a specific task.

Weinshall and Raveh [241] identify three basic managerial structures: entrepreneurial, functional, and decentralized. The *entrepreneurial structure* is typical of an organization recently created, fairly small, and

growing fast. It must be managed in an informal and centralized fashion in order to survive. Everything is centered on one person, the entrepreneur who created the enterprise. When organizations grow beyond a certain size, they must go through a major transformation. An entire set of rules by which the work is managed and carried out needs to be formalized, "in order to cope with the growing quantities of product and services, their variety, and the complexity of the organization" [241, p. 55]. This is called the *functional structure*. The chief executive officer directly controls the various functional heads, such as the production manager, the marketing manager, the sales manager, etc. The formalized and centralized nature of the functional structure must, at some point, give place to the *decentralized structure*. As a result of expansion, the number of managers (people) grows far beyond the number that can efficiently report to a single person. At this stage, the organization must slow down its growth and introduce a new formal and decentralized structure, organized by product/service line or by geographical area.

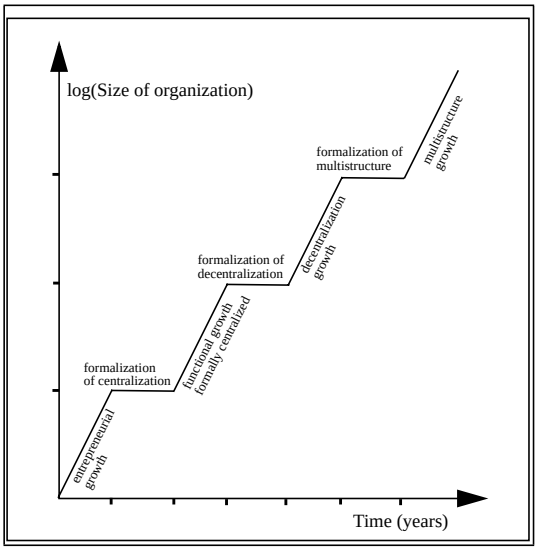

Fig. 5. The effect of size on the enterprise structure (adapted from Weinshall and Raveh [241, pp. 56–57])

These three structures are uniform, in the sense that "all subordinates of the chief executive are structured either in an entrepreneurial, or a functional, or a product line or area structure" [241, p.188]. Beyond a certain size, this uniformity cannot be maintained: the decentralized structure needs to change into a *multistructure*, i.e. a federated managerial structure where "different building blocks may be combined into different kinds of structures" [241, p. 189]. The Japanese, according to Weinshall and Raveh, were the first to operate their large organizations in multistructures, in a type of organization known as *zaibatsu*. The multistructure is inherently flexible, in that it enables changes in the composition of the federated basic structures. This natural evolution as enterprises grow from the entrepreneurial structure to the multistructure is depicted in Fig. 5. The actual values of the time on the x axis and the logarithm of the size of the organization on the y axis depend on the sector of activity of the enterprise, and change dramatically from one type of industry to another.

To conclude with enterprise management, let us take a look at the *evolutions and revolutions cycle* depicted in Fig. 6, an evolution trend that Greiner identified back in 1972. In retrospect, it is amazing to see how this cycle, devised for enterprise management, suits NSM well. If the first three phases look similar to those identified by Weinshall and Raveh, the last two, coordination and collaboration, are incredibly visionary and predicted three decades ago what intelligent agents are now striving to achieve in NSM! It is also interesting to notice that the last crisis is left as unknown. Would the combination of hierarchical and cooperative paradigms be the ultimate solution? Or was the idea of collaboration so new in the enterprise world that Greiner, lacking hard evidence, did not want to speculate on what could go wrong then?

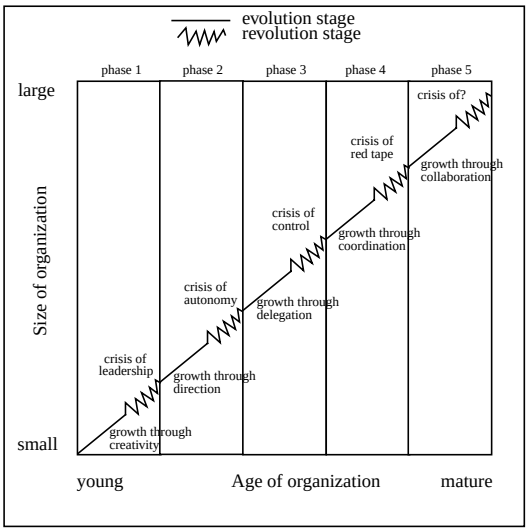

Fig. 6. The five phases of organization development (adapted from Greiner [91, p. 39])

3.2.1.2 Delegation granularity

What do Mullins, Greiner, and Weinshall & Raveh tell us that could apply to NSM? First, all the management paradigms they consider are hierarchical, except for the last two phases described by Greiner, which are characterized by a mix of hierarchical and cooperative paradigms. Likewise, most distributed NSM paradigms are hierarchical today, and cooperative paradigms have only just begun to appear in hybrid structures similar to Weinshall and Raveh's multistructures. Second, delegation schemes should evolve as enterprises grow in size, otherwise they become inefficient. Similarly, distributed NSM should rely on different distributed NSM paradigms as networks grow in size and complexity. In this respect, the recent explosion of new distributed paradigms seems justified, because networks have grown by at least an order of magnitude in terms of size and complexity since the SNMPv1 and OSI management architectures were devised.

Third, if there are many ways of dividing up enterprise organization structures, depending on the granularity of the analysis, most authors agree with Weinshall and Raveh that they all coalesce in three broad types: by function, by product or service, and by geographical area. We can see some similarities here between enterprise management and distributed NSM. The division of management domains by geographical area, for example, makes sense in both worlds. But there are clear discrepancies too, since the basic entities are people in one case, and machines or programs in the other. The division by function only makes sense in the enterprise world. For instance, it takes many years for a person to become an expert in accounting or electrical engineering, and an accountant cannot be turned into an engineer overnight; conversely, a computer can be equipped with new competencies in a matter of minutes or hours, by simply transferring a few programs. We will show next that Mullins's federal decentralization and Weinshall and Raveh's decentralized structure map onto our *delegation by domain* scheme in distributed NSM, while Mullins's functional decentralization and Weinshall and Raveh's functional structure map onto our *delegation by task* scheme.

The fourth and most important thing we can learn from enterprise management is this common evolution trend, of which Greiner and Weinshall & Raveh give two different, but compatible, versions. There is a natural evolution of companies from centralized structures to decentralized ones, and from lightly decentralized structures (organized by function) to more decentralized ones with independent units (organized by product/service or by geographical area), to even more decentralized structures based on federation or cooperation. In NSM terms, these four stages map easily onto the different types presented in our simple taxonomy. The evolution which occurred in enterprise management over the 20th century suggests that the same evolution may take place in NSM in the next decade or so. The time scale may be different, but the evolutionary trend toward more distributed and cooperative management is the same.

Delegation by domain vs. delegation by task

How do the eight types of delegation identified by Mullins translate into NSM terms? We saw that delegation by geographical domain applies equally well to both worlds, but what about the other types? In 1994, Boutaba [29] identified a number of criteria to define domains in NSM. Resources are grouped into domains when they share a common feature. This may be the organizational structure (same department, same team), the geographical location, access permissions (resources accessible to a user, a group of users, or everybody), the type of resource (same vendor, same management protocol), the functionality of the resource (printer, mail system), or the systems management functional area [48]. Some items in this list resemble Mullins's types, although they were made in very different contexts. But both of these lists are far too detailed for our taxonomy. In NSM, we propose to group all possible delegation policies in just two types: delegation by domain and delegation by task.

Delegation by domain relies on static tasks. The manager at level (N) assumes that the manager at level (N+1) knows all of the management tasks to be completed within its domain (N=1 for the top-level manager, N=2,3,4... for the mid-level managers). In today's networks, delegation by domain typically translates into delegation by geographical domain, to manage geographically dispersed enterprises. For instance, let us suppose that the headquarters of a multinational organization are located in Sydney, Australia. This enterprise cannot afford to manage its large subsidiaries in the USA, Asia, or Europe over expensive and relatively slow transcontinental WAN links. Let us consider its European subsidiary, located in Geneva, Switzerland. The manager in Sydney delegates the entire management of the Swiss subsidiary to the manager located in Geneva, and expects it not to report that a local printer goes down, but to report that the number of errors per minute exceeds a given threshold on the Switzerland-Australia WAN link. The point here is that the Australian manager does not tell the Swiss manager what to report; instead, the Swiss manager is expected to make this decision by itself. In practice, this translates into a human being, the LAN administrator, hard-coding in the Swiss manager what to report back to Sydney and how to manage the rest of the LAN. There is no mechanism for the Australian manager to alter the way the Swiss manager manages its domain: it is a *carte blanche* type of delegation, whereby the Geneva-based manager has total control over its own LAN. Network management is not automated, and there is no way for the Australian network administrator to enforce a management policy over all of its subsidiaries. Clearly, these are serious limitations.

Delegation by task, conversely, offers a finer-grained vision at level (N) of the management processing occurring at level (N+1). As a result, the manager at level (N) can see the different tasks at level (N+1), as well as other tasks of its peers at level (N). Tasks need no longer be static and hard-coded in every manager: they can also be modified dynamically. This idea was first applied to NSM with Management by Delegation, as we saw in Section 3.1.6.1. Goldszmidt departed from the well-established notion of static tasks underlying the centralized paradigm, and introduced the notion of dynamic tasks, transferable from the manager to its subordinate agents. This paradigm was later generalized to transfer dynamic tasks from a manager at level (N) to a manager at level (N+1).

Microtasks vs. macrotasks

A manager at level (N) has several ways of driving a subordinate at level (N+1). With traditional approaches such as SNMPv1, the basic unit in the manager-agent dialog is the protocol primitive: the manager issues a series of `get` and `set` requests to the agent. The data manipulated are MIB variables, which are statically defined when the MIB is designed. With large MIBs or large networks, this leads to the micro-management syndrome [88], which entails significant network overhead and a poor use of the resources of the managers, managed devices, and managed systems.

Recent approaches avoid this syndrome by splitting the management application into many different units, or *tasks*, and by distributing these tasks over a large number of managers and agents, while still letting the manager at level (N) be in control of what subordinates at level (N+1) do. The underlying mechanism of this distribution is independent of the tasks being delegated; it can rely on program transfer, message passing, Remote Procedure Calls (RPCs), etc. The focal point for the management application is the granularity of the

delegation, that is, the way the work is divided. Clearly, there is a wide spectrum of task complexities, ranging from the mere addition of two MIB variables to the entire management of an ATM switch. We propose to distinguish only two levels in our enhanced taxonomy: microtasks and macrotasks.

A *microtask* (μ-task) simply performs preprocessing on static MIB variables, typically to compute statistics. It is the simplest way of managing site-specific, customized variables. There is no value in these data *per se*, which still need to be aggregated by the manager one level up. If contact with the manager is lost, statistics are still gathered, but there is no way for the subordinate to take corrective action on its own. In the case of a *macrotask* (M-task), the entire control over an entity is delegated. A macrotask can automatically reset a network device, or build an entire daily report, etc. If contact is lost with the manager one level up, corrective actions can be automatically undertaken.

3.2.2 Other Criteria for Our Enhanced Taxonomy

In previous work [130], we studied the features that designers of management applications want from strongly distributed management paradigms. We identified a number of criteria and showed that in addition to interoperability and scalability, which are already addressed by weakly distributed management paradigms, the two most critical criteria for designers are (i) the semantic richness of the information model, and (ii) the degree of automation of management allowed by a paradigm.

3.2.2.1 Semantic Richness of the Information Model

The semantic richness of the information model of a management application is an indication of the expressive power of the abstractions used in this model. It measures the facility for designers of management applications to specify a task to be executed by a manager or an agent. The higher the level of abstraction used to model a management application, the higher the semantic richness of the information model, and the easier it is for someone to build and design a management application.

It is well known in cognitive sciences that computers can easily be programmed to deal with low-level abstractions, but cannot easily manipulate higher level concepts; people, conversely, find it easier to think at a high level of abstraction, but are easily overwhelmed by too many low-level concepts. This is also true for NSM administrators, particularly when they design large or complex management applications. Unfortunately, management architectures have traditionally offered fairly poor APIs, thereby constraining designers to model management applications with low-level abstractions.

In this section, we show that this limitation has been addressed recently by some of the new management paradigms. Today, designers of management applications have the choice among three types of abstractions to build an information model:

- managed objects, offering low-level abstractions
- computational objects, offering high-level abstractions
- goals, offering very high-level abstractions

Let us review these three types of abstractions. We will introduce and compare the concepts of protocol API and programmatic API, and will identify a new criterion for our enhanced taxonomy: the degree of specification of a task.

Managed objects

Both the SNMP and the OSI management architectures offer a *protocol API*. In these architectures, there is a one-to-one mapping between the communication model and the information model, to use the ISO/ITU-T terminology [50]. The semantics offered to the designer of a management application—that is, the kind of entities and actions that can be defined in the information model and constitute the building blocks of a management application—are constrained by the communication protocol primitives used underneath. The

protocol is not transparent to the application; this breaks a well-established rule in software engineering. For instance, with the APIs available in the different SNMP architectures, administrators have to think in terms of SNMP `get` and `set` when they write a management application (e.g., with Perl [238] or Tcl [160] scripts).

We call this the *managed-object* approach, as both the IETF and the ISO use this term to describe a unit of the information model in their respective management architectures. All technologies based on centralized or weakly distributed hierarchical paradigms share this approach. When a management application is designed with managed objects, a protocol is automatically imposed; the managed objects must live in full-blown agents (in the case of TMN, these agents need to implement a large part of the OSI stack, including especially CMIP and CMIS); and the manager-agent style of communication is imposed. These are very strong constraints imposed on management-application designers.

We stress that the identity between the communication and information models has nothing to do with the protocols themselves. It is implicit in the SNMP and OSI management architectures. The limitations entailed by this approach reflect on the apparent limitations of some technologies. In the recent past, some Java-based management platforms used a simple port of the very basic SNMPv1 API developed at Carnegie Mellon University in the early 1990s (the `snmpget` and `snmpset` C programs). But nothing inherent in Web-based management prevents Java-based technologies from using richer APIs that deal with higher-level abstractions.

To address this, some people developed richer APIs to managed objects. One of the authors proposed an API [254] based on the Structured Query Language (SQL) to leverage the natural mapping between SNMP tables and tables found in relational databases. One advantage of making SQL queries from the management application, rather than SNMP `get`'s, is that table handling in SQL is less tricky than SNMP-table handling. (SNMPv1 is fairly poor at dealing with sparse tables.) But SQL-based APIs have not encountered much success so far, probably because relational databases are too slow and too demanding in terms of resources to become widely available in agents.

Computational objects

Protocol APIs for distributed systems are based on ideas that began to be criticized in the late 1970s and early 1980s, in particular by the software-engineering research community which was then promoting the concept of *objects*. Since the mid-1980s, this community has been advocating the use of *programmatic APIs* instead, which have been one of the selling points of the object-oriented paradigm for distributed systems. With such APIs, any object belonging to a distributed system is defined by the interface it offers to other objects. The distributed object model is independent of the communication protocol: it only defines a programmatic interface between the invoker and the operations (methods) supported by the invoked object. This programmatic API relies on a protocol at the engineering level, but this protocol is completely transparent to the management-application designer.

We call this the *computational-object* approach, with reference to the terminology used by the ISO for ODP and ODMA. In this approach, designers of management applications can use class libraries that offer high-level views of network devices and systems. Few constraints are imposed on the design: objects may be distributed anywhere, they need not live in the specific agents that implement specific protocol stacks. The only mandatory stack is the one that implements the distributed processing environment. No specific organizational model is imposed or assumed. The management application relies solely on object-to-object communication. The administrator may define site-specific classes and use them in conjunction with libraries of classes that implement standard MIBs.

The computational-object approach is the main strength of many recent management technologies, most notably distributed object technologies. In NSM, it accounts to a large extent for the recent success of Java in the IP world. Strangely enough, it is not responsible for the even greater success of CORBA in the telecommunications world: CORBA has mostly relied on a managed-object approach so far. Telephone switches are very complex to manage, considerably more than IP routers. The sole fact that, with CORBA, the millions of lines of code necessary to manage a switch could be written in parallel by many independent programmers, from

different companies, in different languages, was in itself a blessing. And dealing only with well-known managed objects was a guarantee of interoperability. Even though CORBA objects could have conveyed higher levels of abstraction than OSI managed objects, the telecommunications industry has been happy so far to simply translate managed objects into CORBA objects. Will the success of Java-based management in the IP world suggest new ways of using CORBA in telecommunications in the future?

Goals

The third type of abstraction that may be used in information models is the *goal*. In Section 3.1.7, we saw that cooperative paradigms are goal-oriented. The management application is split into tasks, which are modeled with very high-level abstractions and partially specified with goals. Once these goals have been sent by the manager to the agent, it is up to the agent to work out how to achieve these goals.

This approach is fundamentally different from the one taken by weakly or strongly distributed hierarchical paradigms, whereby the management application is broken down into fully specified tasks. Whether the implementation of the task relies on calls to communication-protocol primitives or method calls on objects, the agent is given by the manager a step-by-step *modus operandi* to achieve its task. With strongly distributed cooperative paradigms, it is not.

Goals may be specified via a programmatic API, a protocol API, or both. They do not require an object-oriented distributed system to be used underneath. But the coupling of agents and objects looks promising in NSM. Knapik and Johnson [120] describe different styles of communication between intelligent agents: object-oriented agents rely on remote method calls, whereas plain agents rely on communication languages such as KQML [75]. The primitives (performatives) of KQML are considerably richer than those of SNMP and CMIP, thus goals are less limited by protocol APIs than managed objects.

To date, goals represent the highest level of abstraction available to management-application designers. They rely on complex technologies known as *intelligent agents* (see Section 3.1.7), which often support some kind of inference engine and pattern learning, and are generally not available on managed systems or network equipment. There is still a large market for simpler technologies that support computational objects, or even simpler technologies that support only managed objects. But goals are a type of abstraction that makes it possible to manage very complex networks, systems or services, for which simpler abstractions are not suited. They are particularly well suited to support negotiation, load balancing, or resource usage optimization.

Degree of specification of a task

Managed objects and computational objects rely on fully specified tasks, whereas goals rely on partially specified tasks. In other words, the semantic richness of the information model and the degree of specification of a task are tightly coupled. We decided to retain the latter as a criterion for our enhanced taxonomy, because it shows two very different ways of specifying tasks in a management application. But we must keep in mind that these two criteria are not independent.

management-application unit (= information model abstraction)	managed object	computational object	goal
abstraction level	low	high	very high
where does it live?	MIB	object	intelligent agent
how do we access it from the management-application code?	management protocol primitives (SNMP, CMIP, HTTP)	method call	1) agent communication language primitive (KQML) 2) method call
degree of specification	full	full	partial

Table 2. Semantic richness of the information model

3.2.2.2 Degree of Automation of Management

Until a few years ago, the main drive behind the automation of management was to relieve network and systems support staff from the burden of constantly staring at GUIs and of solving problems manually as they occur. As systems and networks grow yearly in size and complexity, administrators become increasingly eager to automate their management: *ad hoc* manual management is not sufficient anymore. Yemini claimed a few years ago that "management should pursue flexible decentralization of responsibilities to devices and maximal automation of management functions through application software" [250, p. 28].

Today, the need for more automation of management is also determined by two factors: the deregulation of the telecommunications industry worldwide and the explosion of new services offered to end-users. First, there is fierce competition in the telecommunications market. Monopolies (or near monopolies) have given way to a plethora of competing network operators, service providers, service traders, content providers, etc. So any service provision today is likely to cross several networks, managed by different companies, with equipment from several suppliers [3]. Second, more and more services are offered: mobile telephony, electronic commerce, video on demand, videoconference, teleteaching, telemedicine, etc. Videoconferences, for example, used to be booked by fax on an *ad hoc* basis. End-users would contact support staff several days in advance; support staff would fax the single provider on the market (the local network operator); they would receive a reservation confirmation and an invoice within a day or so, sometimes less; and finally, they would inform the end-user that the booking had been made. This process was time consuming, very inefficient, and error prone. Today, end-users want to deal directly with a service trader via a user-friendly GUI, get the best possible deal for a videoconference scheduled at most a couple of hours in advance, and make an electronic transaction with a click of a mouse. Tomorrow, videoconferences will not be scheduled but will be provisioned immediately, like telephone today. Such demands are much more stringent than they used to be, and require considerably more work than mere faxes. As the number of such transactions grows within enterprises (from once a month to once an hour to once a second), and as the demands become more stringent, manual handling becomes less often an option. Service management must be automated to offer the online GUI that the end-user expects. Network management also has to be automated, e.g., to handle resource reservations and potential rerouting. Eventually, systems management must in turn be automated, e.g., to provide for automatic failover (*hot stand-by*) for video-on-demand servers.

As we show in Fig. 7, microtasks poorly automate distributed NSM; but macrotasks are very good at it, because they enable remote agents to take corrective actions independently from the manager. Intelligent agents are typically used in negotiation, e.g., to get the best deal for a cross-Atlantic videoconference from competing service providers. But they are also good at dealing with the dependencies between service, network, and systems management. To summarize the need for automation, the larger and the more complex the networks or the systems, the more automated the management application should be.

3.2.3 Synthetic Diagram

Our enhanced taxonomy is now completed. It consists of the four criteria presented in the previous sections:

- delegation granularity
- semantic richness of the information model
- degree of automation of management
- degree of specification of a task

These criteria are not independent. The semantic richness is closely linked to the degree of specification of a task, as is the delegation granularity to the degree of automation.

Note that in Fig. 7, the axes take discrete values, not continuous values. In other words, the relative placement of different paradigms in the same quadrant is meaningless.

Note that we do not list all existing technologies in our enhanced taxonomy. For strongly distributed management, for instance, only paradigms are depicted. The reason for this choice is threefold. First, we want to keep this taxonomy readable. Second, technologies evolve so quickly, and this market is currently so active, that any such effort would be doomed to fail: such information would be obsolete as soon as it is published. Java, for instance, has blurred the boundaries between mobile code and distributed objects in the recent past. Third, we believe that the criteria we selected and presented are easy to understand, and that potential users of such technologies should be able to decide where to locate a given release of a given technology in Fig. 7, based on a short technical description of it.

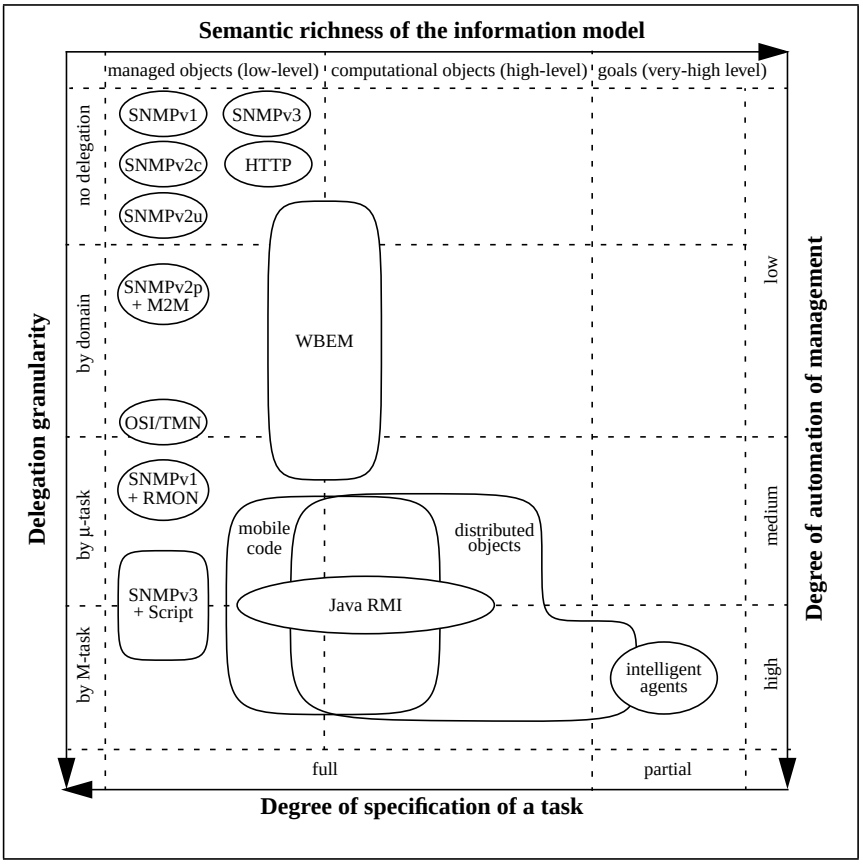

Fig. 7. Enhanced taxonomy of NSM paradigms

By counting the quadrants that are populated in Fig. 7, we see that our enhanced taxonomy consists of nine types:

- no delegation with low-level semantics;
- no delegation with high-level semantics;
- delegation by domain with low-level semantics;
- delegation by domain with high-level semantics;
- delegation by microtask with low-level semantics;
- delegation by microtask with high-level semantics;
- delegation by macrotask with low-level semantics;
- delegation by macrotask with high-level semantics; and
- delegation by macrotask with very high-level semantics.

3.3 Summary

In this chapter, we proposed two taxonomies to classify all major management paradigms and technologies available to date for managing IP networks and systems. In our *simple taxonomy*, all NSM technologies were classified according to their underlying organizational model. We grouped them into four different types of NSM paradigms: (i) centralized paradigms, (ii) weakly distributed hierarchical paradigms, (iii) strongly distributed hierarchical paradigms, and (iv) strongly distributed cooperative paradigms. Faced with dozens of commercial or prototype NSM technologies on the market today, with new ones appearing every month, designers of management applications run the risk of being overwhelmed by the abundance of choice. With this simple taxonomy, they now have a simple tool to find out quickly which management paradigm lays behind a given technology.

The purpose of our *enhanced taxonomy* was to go beyond the sole understanding of the management paradigm by providing criteria to actually select a paradigm first, and then a technology. To this end, we identified four criteria: (i) the granularity at which the delegation process takes place (by domain, by microtask, or by macrotask); (ii) the semantics of the information model (managed object, computational object, or goal); (iii) the degree of automation of management (high, medium, or low); and (iv) the degree of specification of a task (full or partial). This enhanced taxonomy complements the previous by being more practical. It gives some arguments for designers of management applications to select one paradigm rather than another, based on the issues they face during the analysis and design phases.

Chapter 4

ANALYSIS OF THE SOLUTION SPACE

Our objective in this chapter is to analyze the solution space described in Chapter 3 and draw some high-level conclusions for our management architecture. In particular, we explain why we selected Web technologies and weakly distributed hierarchical management in our proposal.

This chapter is organized as follows. In Section 4.1, we show that there is no single winner among all the solutions described in Chapter 3, and that different solutions are best suited for different management tasks. In Section 4.2, we highlight that it is important that administrators do not focus immediately on selecting a technology. In Section 4.3, we perform a reality check on the different technologies and paradigms investigated in Chapter 3. The issues of support and technical maturity lead us to eliminate cooperative management. In Section 4.4, we explain why the *my-middleware-is-better-than yours* syndrome leads us to rule out the solutions based on distributed object-oriented middleware. In Section 4.5, we investigate why mobile code is not ready yet, and why we need to prepare for its future integration. In Section 4.6, we come to the conclusion that the distribution of NSM should rely on a weakly distributed hierarchical management paradigm. In Section 4.7, we decide to adopt Web-based management for the next management cycle. Finally, we summarize this chapter in Section 4.8.

4.1 No Win-Win Solution

The first conclusion that can be drawn from our taxonomies is that there is no win-win solution for the next management cycle. Depending on the size and complexity of the network, depending on the nature and complexity of the system, application, service, or policy to manage, some paradigms are better suited than others. This diversity gives us flexibility. Certain paradigms, e.g. mobile code, encompass a large number of technologies and can be further subdivided into several paradigms. This yields a wide variety of fine-grained designs. Designers should thus no longer feel constrained when they model management applications. Among all available paradigms, they should select the one that allows them to model the problem at hand in the most natural way. The days of protocol APIs are over, when the focus was on the nuts and bolts of NSM: management applications can now rely on programmatic APIs, where the focus is on software development and user friendliness.

To illustrate that different solutions are best suited to different tasks, let us study a simple scenario that illustrates the growing management needs of an enterprise over time. In this scenario, we are concerned only with the design qualities of the management paradigms. Their practicality will be investigated in subsequent sections.

In a small company, to manage a small LAN or a distributed system comprising a dozen machines or so, there is no need for an expensive technology offering a high level of automation with computational objects, or even goals: a less expensive solution based on managed objects and microtasks is sufficient. Centralized management is suitable in this case.

Over time, this enterprise develops and opens branches abroad. It is now a geographically dispersed midsize enterprise, but still has fairly simple needs (data network, no multimedia services). A weakly distributed hierarchical technology is well suited. The required degree of automation is medium, and managed objects are sufficient to deal with simple needs. RMON is a good candidate for monitoring and performance management. When the bandwidth of the WAN links connecting the main office to the remote offices (or, at a larger scale, the headquarters to the subsidiaries) becomes too expensive, it is time to migrate from centralized to weakly distributed hierarchical management: each subsidiary is equipped with a local management platform.

As people in this enterprise begin using multimedia services on a more regular basis, there comes a time when the semi-automated handling of reservations and bandwidth allocations is no longer an option: a higher degree of automation is required. Inexpensive, distributed object technologies usually suffice. But some cases can be better handled by intelligent agents. Typically, when confronted by a new service, an intelligent agent can make a decision automatically, without any human intervention, based on what it learned in the past with other services. For instance, it can make a trade-off between the (usually too high) requirements set by a new end-user and the cost that the user's department is willing to pay for reserving network bandwidth. Manual intervention by an administrator is performed afterward only when the trade-off turns out to be poor.

As this enterprise develops, the number of entities to manage grows so large that the management application becomes too complex and cumbersome. It becomes difficult to use, awkward to modify, and any change may cause a new problem due to unforeseen side-effects. The semantic richness of the information model is too poor: managed objects have become inadequate. Even for simple day-to-day management tasks, it is now time to use computational objects instead. As new services are adopted by users, new intelligent agents are added on an *ad hoc* basis.

The midsize company has now become a large enterprise. Intelligent agents are no longer restricted to dealing with novel services. They can be used for complex tasks such as distributed pattern learning or data mining. For example, they can dynamically learn the peak and slack hours of a VPN overlaying an ATM network, and automatically readjust the bandwidth rented from the service provider in order to reduce the running costs for the enterprise. Intelligent agents can also learn dynamically at what time of the day voice-over-IP and data traffic must be routed over different WAN links to guarantee the required QoS, and when it becomes possible to route them through the same WAN links (slack hours); this, too, allows the enterprise to temporarily rent less bandwidth and thus to reduce its bill.

Later, this enterprise is bought by a large multinational with tens or hundreds of thousands of managed systems and devices. The degree of automation of management then becomes critical. This time, day-to-day NSM should entirely rely on distributed objects, and managed objects should be banned from interactive APIs. There is no way the operators can find the time to go down to instrumentation types of abstraction to understand or correct a problem: they must have only high-level decisions to make. Clearly, if day-to-day NSM is already based on distributed objects prior to the merger, the integration of a smaller management application with a larger one is considerably easier.

Finally, in the large multinational, the number of requests for high-level services (e.g., multimedia services requiring bandwidth reservation) gradually increases. So does the diversity of the services that are used on a daily basis. The number of specialized intelligent agents is now high, and it no longer makes sense to segment this knowledge. This calls for elaborate multi-agent systems, whereby a large number of intelligent agents

cooperate in order to optimize network-bandwidth usage and reduce the number of reservations that could have, but were not, granted. Note that no one has already tested such systems on a large scale in NSM.

We hope to have highlighted one important thing in this series of examples: the relative weights given by the administrator to the different selection criteria of our enhanced taxonomy give a clear indication of the best suited management paradigm.

4.2 Do Not Focus on the Technology Immediately

The second conclusion that can be derived from our enhanced taxonomy (see Fig. 7, p. 58) is that there is not a unique relationship between a quadrant, a management paradigm, and a technology. Several paradigms span over multiple quadrants and therefore offer different degrees of automation, or different levels of semantic richness of the information model. Similarly, a single technology can support different paradigms, and can thus offer different degrees of automation, different delegation granularities, different degrees of specification of a task, or different levels of semantic richness.

The practical consequence of this remark is that selecting a technology solves only a fraction of the problem—although many administrators today focus mostly on this step. Initially, the most important is to determine the criteria of our enhanced taxonomy that are the most relevant to a specific enterprise. Once this has been done, the management paradigm can easily be derived from our enhanced taxonomy. Once these two steps have been made, it is time to select a technology with our simple taxonomy.

This is illustrated by the following example. Convinced by an advertising campaign, an administrator decides that mobile code is the solution to manage his/her network. Before delving into the design of a new, powerful management application, he/she decides to investigate the market of mobile-code technologies. All vendors claim to sell the best product, so what technology should he/she choose? With our enhanced taxonomy, our administrator can see at a glance that under the same name, he/she can actually purchase four very different types of technologies, because mobile code spans over four quadrants. Some technologies offer low-level semantics, others high-level semantics; some offer a high degree of automation of management, others a medium degree; some support delegation by microtask, others by macrotask. The administrator also sees immediately that mobile code encompasses three finer-grained paradigms: remote evaluation, code on demand, and mobile agents. In the end, our taxonomies allowed this administrator to realize that mobile code is multifaceted, and to choose the technology offering the best value-for-money according to the relative weights that he/she gave to the four selection criteria of our enhanced taxonomy.

4.3 Reality Check: Support and Technical Maturity

In our two taxonomies, we classify and compare many different technologies and paradigms that can be used in NSM. The fact that these potential solutions are all presented side by side does not imply that they are equally ready to be used in industry, let alone appropriate for a large market such as NSM. We structured the solution space in technical terms, based on the elegance and promises of the designs rather than the feasibility of the solutions. But in order to select an appropriate solution for the next management cycle, it is important to make candidate solutions go through a reality check. Among the technologies we described in the solution space, many lack proper support and are confined to the research community. Some do not even go beyond the proof of concept... To deploy a solution in a production environment, as opposed to prototyping in a testing environment, it does not suffice that the design be technically appealing: the technology itself must be well tested and well supported. In addition to its technical qualities, the solution retained for the next management cycle should therefore exhibit two important qualities: support and maturity.

Support includes commercial and technical support. *Commercial support* is necessary in almost all production environments. If a management application does not work, the only guarantee of the customer is that its supplier be legally bound to delivering quality software. Many solutions presented in Chapter 3 are not

commercially supported, because they are research prototypes simply demonstrating a good idea. *Technical support* is equally important; it is primarily about maintenance. In case a major problem occurs, the customer must be able to receive technical support by its supplier's support team. This, too, is mandatory in a production environment. As far as the management architecture is concerned, we should therefore select a solution where commercial and technical support are available.

To put it simply, *technical maturity* means that a solution is not appropriate until it has been sufficiently debugged by others. If the provider has not yet acquired enough experience by working with other customers, the management platform is not really a commercial software, it is rather a prototype for beta-test field sites. And many enterprises do not have the skills, manpower, and organization to enroll as a beta-test field site. An important aspect of maturity is that it takes time before a technical support team becomes efficient: the staff needs to be trained through a variety of problems at many sites and a lot of trial-and-error stumbling. Consequently, the market for the paradigm that we select for the next management cycle should be a commodity market: it would be too hazardous to select a technology that is still confined to a niche market.

These two criteria, support and technical maturity, lead us to eliminate all the technologies that pertain to cooperative management. The DAI community is still at an early stage of research with respect to intelligent agents and multi-agent systems. Many technological problems are still unsolved. For instance, it is not clear what language should be used to communicate between agents (KQML, ACL, etc.), or what format should be used by the agents to exchange "knowledge" (KIF, etc.). In the specific case of NSM, it is not clear at all what ontology should be used for managing IP networks and systems. Worse, it is not even clear what properties should be exhibited by a "good" agent-communication language, a "good" ontology, etc. All of these problems are very exciting to study, but it will take years before the DAI community builds the *corpus* of knowledge necessary for the cooperative-management paradigm to be usable in application domains. We are still very far from a commodity market. Until this level of maturity is reached, it remains unrealistic to base the management architecture of the next management cycle on intelligent agents, multi-agent systems, or more generally on the cooperative management paradigm. The slow pace at which the FIPA consortium is progressing in the telecommunications application domain is an indication that these problems are not yet well understood and not easy to solve.

In short, WIMA should be based on a management paradigm for which there are commodity technologies that are both technically mature and well supported.

4.4 The *My-Middleware-Is-Better-Than-Yours* Syndrome

There is a fundamental problem with using object-oriented middleware in NSM. This problem is well pictured by the following analogy with viruses:

> The *my-middleware-is-better-than-yours* syndrome is a disease that has spread in the software industry in the second half of the 1990s. It is known to have already infected most middleware platforms on the market, be they based on open standards such as CORBA and Java, or on proprietary solutions such as Microsoft's DCOM. The symptoms of this syndrome are common to all well-known forms of infection:
>
> - Only consenting individuals can be infected.
> - Once infected by a middleware, an individual is obliged to use it to communicate with all other individuals because it immediately loses all its previous communication skills.
> - Once infected, an individual takes time to learn to live with a middleware.
> - Unlike what happens with most diseases, an individual can replace its middleware with another at any time. But because of the previous point, this should be avoided.
> - Once an individual is infected, its middleware quickly takes up most of its CPU, memory, and disk resources. A middleware likes to have a large footprint on individuals.
> - A middleware is very exclusive: to communicate with an infected individual, you must have

already caught the same disease. This trait is the sheer result of adaptation through darwinian selection: the disease spreads by consent, so there must be an incentive for individuals to voluntarily get infected!

- A middleware loves gambling: you must be infected *before* you know what middleware will be supported by the majority of the population in the future.
- A middleware is antisocial: it prevents individuals from communicating with a large proportion of the population (that is, the individuals infected by another middleware).
- A middleware is merciful: if an individual cannot get infected but really needs to communicate with an infected individual, it can use a translation service called *gateway.* This service is slow, cumbersome to use, and loses some semantics. But how could you possibly argue once you have been pardoned...

In other words, middleware packages, especially in the object-oriented market, have largely diverged from their initial objective: to allow all machines to interoperate in an open way, with a high level of semantics, via standard interfaces, with a standard protocol, through an object-oriented DPE. Due to diverging commercial interests, the middleware market has been segmented, with different vendors or consortia trying to justify why their middleware is better than the others. Today, when you choose a middleware package, you must choose your camp: CORBA, Java, DCOM, etc. And of course, you can make the "wrong" choice—that is, the market can coalesce around another middleware two years after you have invested a fortune in the "wrong" middleware. We believe that this situation will last a long time: there will be no winner in the middleware battle, at least for many years. As a result, a management architecture cannot reasonably rely on one particular middleware to be available on all network devices and systems worldwide.

The two messages that we try to capture in the catchphrase "the *my-middleware-is-better-than-yours* syndrome" are (i) the fight between vendors who think they know better, and (ii) the risk factor for customers because they must choose their camp.

Another problem shared by most middleware platforms is their large footprint on the agent. In the telecom world, this is not very important because the agents are usually not resource bounded: the cost of adding more memory or processing power to a $1,000,000 telephone switch is negligible. In Section 3.1.6.2, we mentioned the growing acceptance of CORBA as the standard object-oriented middleware in the telecom world. Note that in the telecom world, agents can even afford to support several DPEs simultaneously (although they are not always easy to integrate).

In the IP world, conversely, the footprint of CORBA or J2EE (Java 2 Enterprise Edition, based on EJBs) on an agent is a no-no. Even J2SE (Java 2 Standard Edition, based on standard Java) or a mere JVM are not always an option. In the IP world, most agents have limited resources available to management. The cost of the middleware is also an important issue in the IP world: when a simple network device costs less than $100, you do not want to double its price just to pay for the CORBA licence!

In conclusion, for WIMA, high-profile, object-oriented middleware platforms such as CORBA, Java, or DCOM are not an option. We must limit ourselves to a lower-profile solution, with a low footprint and a low cost per agent.

4.5 Mobile Code and Security

Mobile code has received a lot of attention in the recent years, as we explained in Section 3.1.6.1. So far, in NSM, we have not really seen a good reason for using mobile code instead of more standard technologies. On a case-by-case basis, remote evaluation, code on demand, or mobile agents can perform better than the client-server model that underlies most other solutions. But we still have not found a management task that is always better performed by mobile code.

The situation is very different in service management, where mobile code has proved to be very useful for dynamic service provision, especially in the area of *active services* [36]. It is currently considered *the* way to go by the people who believe that we cannot possibly have the same middleware on all machines worldwide.

The main problem with mobile code is security. This is the main obstacle to its wide deployment. There are actually two different problems: we must secure the host from a malicious code [179] and the code from a malicious host [116]. In integrated management, the first problem is the one that is generally considered today. But both need to be solved before we can use mobile code in a new, general-purpose management architecture. These security problems have proved harder to solve than people initially expected. For a good introduction to the range of issues at stake, see Vigna [229].

Because these security issues are still unsolved, it is too risky to base the next management cycle on this new paradigm. Software engineers first need to understand how to secure mobile code in general. Then, NSM specialists will be in a position to leverage it in management. But since it is very likely that some people will use mobile code for service management in real life, it seems appropriate to devise a management architecture that can easily integrate mobile code technologies as soon as they have been secured.

In WIMA, we did precisely that. Our organizational and communication models do not require mobile code, but we offer a simple means of integrating it with XML.

4.6 Distribution

As we saw in Chapter 3, the way to address the scalability issue in SNMP-based management is to distribute management. In Section 4.3, we ruled out cooperative management. So we are left with two alternatives: weakly and strongly distributed hierarchical management (see p. 40).

In Section 3.1.6, we grouped strongly distributed hierarchical management paradigms into two categories: mobile code and distributed objects. We explained in Section 4.4 that because of the *middleware-is-better-than-yours* syndrome, we must eliminate distributed objects. We cannot expect all agents worldwide to support the same smart middleware. In Section 4.5, we concluded that mobile code is a promising paradigm, but it cannot be used as the basis for the next management cycle until security issues are completely understood and solved—and most of the research community expects this to take some time. Neither distributed objects nor mobile code are a viable option; we therefore eliminate strongly distributed hierarchical management.

In conclusion, WIMA should be based on weakly distributed hierarchical management. At first, this might appear to be a setback, if we compare this solution with the more advanced solutions that were investigated in Chapter 3. But as we describe, chapter after chapter, all the features supported and all the SNMP problems that are solved by our management architecture, we hope to convince the reader of the relevance of this decision.

4.7 Web-Based Management

Our simple taxonomy gives two examples of weakly distributed hierarchical management paradigms: TMN and RMON. TMN is not appropriate in the IP world, as we mentioned already. As for RMON, it is too limited for what we envision to do. We therefore ruled out both of these solutions.

In the end, we decided to devise a new weakly distributed management paradigm that does not appear in our taxonomies. WIMA is a form of *Web-based management*. This expression groups together different types of management paradigms, centralized or distributed, which all share the same characteristic: they use Web technologies. This explains why Web-based management does not explicitly appear in our simple taxonomy: it does not qualify for the criterion retained to build this taxonomy. It does not constitute a type *per se*, but overlaps several types.

The scope of Web-based management will be precisely defined in Chapter 5. This will allow the reader to grasp the WIMA's innovations that are detailed in the subsequent chapters. The distribution aspects of WIMA will be described in Section 6.1.4.4 and Section 6.3.5.

4.8 Summary

In this chapter, we analyzed the solution space presented in Chapter 3. In Section 4.1, we saw that there is no win-win solution: different solutions are best suited to different management tasks. We studied a series of examples where different management paradigms are considered solely with respect to their design elegance, without taking into account any deployment constraints. In Section 4.2, we highlighted a frequent problem in NSM: administrators generally focus on the selection of a technology rather than a management paradigm. Our two taxonomies can help solve this problem. In Section 4.3, we explained that cooperative management technologies do not survive the reality check of support and technical maturity. In Section 4.4, we described the *my-middleware-is-better-than-yours* syndrome and explained that it prevents us from using solutions based on object-oriented middleware, especially CORBA and Java. In Section 4.5, we investigated the case of mobile code and concluded that security issues are not yet understood well enough, thus it is too risky to base a new management architecture on this paradigm. But mobile code will probably be used in service management in the future, e.g. for dynamic service provisioning, so it is a good idea to pave the way for it in NSM. In Section 4.6, we investigated the distribution aspects of the new management architecture and selected weakly distributed hierarchical management. Finally, we concluded in Section 4.7 that the best candidate for the next management cycle is Web-based management.

Chapter 5

STATE OF THE ART IN WEB-BASED MANAGEMENT

Web-based management is an ill-defined concept, because it has greatly evolved over the past years. Many people confuse it with the use of a Web browser to perform a management task—a very simple approach that we call *browser-based management*. The extent of this confusion is such that Harler entitled his book "Web-Based Network Management: Beyond the Browser" [92], to stress that there is more to Web-based management than a mere Web browser. Other people confuse Web-based management with WBEM, because of the considerable marketing hype that plagued WBEM in its early days, before its takeover by the DMTF.

The state of the art presented in this chapter highlights the multiple facets of Web-based management and the increasing richness of the solutions that are based on it. In its full meaning, Web-based management is about leveraging any Web technology in any area covered by integrated management. In this dissertation, we restrict our scope to NSM. The Web technologies considered here are Web browsers, HTTP (HyperText Transfer Protocol), HTML (HyperText Markup Language), XML (eXtensible Markup Language), CGI (Common Gateway Interface), Java applications, Java applets, and Java servlets. Proprietary technologies such as Microsoft's ActiveX and Active Server Pages are not presented here: compared to their standard counterparts, they add no general-purpose features but simply make it easier to work in a specific proprietary environment. For the sake of completeness, we include Java RMI (distributed Java), although we already ruled out this solution for the next management cycle in Section 4.4.

The remainder of this chapter is organized as a taxonomy based on the type of Web technologies supported by the manager and the agent. These types are sorted by increasing order of sophistication and complexity of the manager and the agent. Initially, we describe them in chronological order, as the first uses of Web technologies in NSM were also the simplest. Later, we depart from history because some advanced solutions were proposed several years before less advanced solutions were adopted by the market. In Section 5.1 through Section 5.3, we introduce different techniques called browser-based metamanagement, browser-based management, and three-tier management. They all work with plain SNMP agents and are characterized by a growing sophistication of the manager. In Section 5.4 through Section 5.6, we detail HTTP-based management, XML-based management, and distributed Java-based management. These three types correspond to an increasing sophistication of the agent, and thus of the agent-manager interactions. In Section 5.7, we give references to the ever-growing number of commercial products in this active field. Finally, we summarize this chapter in Section 5.8.

5.1 Browser-Based Metamanagement

In our view, the epoch of Web-based management was is 1993, when NCSA Mosaic began spreading all over the planet [247, pp. 12–14]. This year is often considered the outset of the Web, when it left a closed circle of initiated researchers and reached out for the world. Early Web technologies (that is, Web browsers, HTTP, HTML, and CGI) were used in NSM from almost day one. For instance, the author used them in a test environment in early 1994, and in a production environment from mid-1994. Discussions in Web-related mailing lists and Usenet newsgroups showed that many people were doing similar experiments at the same time. Web technologies did not actually replace SNMP-based management, but rather complemented it for collateral tasks—what we call *metamanagement*. Browser-based metamanagement, which we will now describe, is typical of the early days of Web-based management. It is characterized by the interactive use of a Web browser for performing metamanagement tasks.

5.1.1 Online problem reporting, online usage reports

In a first phase, administrators learned how to use the early Web technologies by developing a few simple HTML pages and CGI programs (scripts or binaries). For instance, by writing a dozen HTML forms and interfacing them with a simple database via CGI programs, one can standardize and automate problem reporting and troubleticketing. This significantly simplifies and decreases the work of calldesks [151]. To give another example, network and systems usage is often monitored with daily, weekly, monthly, and yearly reports providing usage statistics. The goal is to detect in advance that a resource is used close to its capacity, so as to increase this capacity (or reduce the need for this resource) before problems arise. This is a simple and well-known form of proactive management. With Web technologies, instead of printing these usage reports, an administrator can put them online (e.g., in the form of Postscript files) on an open or restricted-access internal Web server, and make them accessible via simple HTML forms with option menus, multiple-choice boxes, etc. Once in electronic format, usage reports can automatically be archived on tape: it is no longer necessary to store print-outs in binders. Clearly, this saves a lot of time and space.

Note that all these tasks could already be done in the past with expensive, proprietary solutions. But Web technologies enabled administrators to do it themselves, simply, at a low cost, in a very short time. And they allowed users to access the same data from a PC, a Mac, a Unix workstation, etc.

5.1.2 Online management procedures, online documentation

In production environments, operators (that is, the staff constantly monitoring the health of the networks and systems, 24 hours a day, 7 days a week) typically follow well-defined management procedures to identify the cause of a problem and correct it. This is mandatory in heterogeneous environments where no single person can know by heart how to troubleshoot thousands of problems on different types of equipment from different vendors. Web technologies gave administrators the opportunity to transfer online all the management procedures that used to be printed on paper and kept in binders, filling up entire shelves. This required the transcription of thousands of files (the originals of the procedures) into HTML pages, but it did not involve writing elaborate applications. The indexing system used by the operators to work their way through the procedures remained the same. But instead of flipping through many binders of procedures, the operators had online access to all the procedures from their own machine. Their navigation through the procedures could be facilitated by using hypertext navigation or search engines. Once the procedures had been converted to HTML, the search engine allowed operators to identify and retrieve a procedure by typing in a few keywords—and this could be implemented in only a few hours with WAIS (Wide-Area Information Servers [203]).

At about the same time, equipment vendors began distributing their documentations in electronic format instead of paper, e.g. on CD-ROM (Compact Disk - Read-Only Memory). By putting these documentations online, administrators made them easier to access. But they also made it possible to embed hyperlinks to the relevant manual pages within the HTML procedures.

5.1.3 Online troubleshooting assistance

In a third phase, administrators helped operators identify the cause of routine problems with online troubleshooting assistance. This involved writing more elaborate applications, with numerous HTML forms and CGI programs. Symptom-driven navigation through HTML forms would help operators narrow down the origin of a network or systems problem to only a few possibilities. Operators were asked simple questions such as the type of equipment, the type of fault, the error message displayed on the console, the severity of the notification received by the management platform, etc. With a few clicks, an operator could save a lot of browsing through the catalogs of procedures.

The integration of online management procedures, online vendor documentations, and online troubleshooting assistance proved to be an important step forward in production environments. The interactive Web interfaces were more user-friendly than the thick binders full of procedures that operators were used to, and online procedures were simpler to update for administrators and operational staff.

5.2 Browser-Based Management

Chronologically, the next step for administrators and operators was to manage agents interactively via a Web browser, what we call *browser-based management*. They did not only execute ancillary management tasks via a browser, they also properly managed equipment. At that time, in the IP world, the agents had only two embedded servers: one for SNMP, and another for `telnet`. Thus, in browser-based management, manager-agent interactions are either based on SNMP or `telnet`.

5.2.1 Troubleshooting scripts executed via the Web browser

Certain management procedures can be automated, because they consist only of commands to execute. They do not require that the operator be physically present in front of the faulty equipment to handle hardware. Some of the troubleshooting operations can easily be coded as scripts. For instance, an operator can use the `snmpset` command to reset an interface via SNMP, or the `ping` command to test the reachability of a host. But other troubleshooting operations are not straightforward to automate, because some client-server programs can only work in interactive mode (e.g., the connection to a remote device via `telnet`). A well-known problem with SNMP is that, for most pieces of equipment, not all the commands supported by the Command-Line Interface (CLI) have an equivalent in an SNMP MIB. In other words, some commands can only be performed from the console or, and this is generally the case for NOCs, via `telnet`. For years, such commands could not be coded in scripts. This problem was solved in 1994[1] by `expect`, a Tcl-based toolkit for automating interactive programs [127]. By wrapping interactive commands with `expect`, administrators could code them as scripts and execute them automatically, in unattended mode. Since then, all the management procedures that do not require hardware handling can be automated.

By sheer coincidence, 1994 is also the year when Web technologies spread out in NSM. A useful feature of CGI is that troubleshooting scripts can easily be turned into CGI programs, or invoked by CGI programs. Coupling HTML pages with troubleshooting CGI programs allowed administrators to automate management procedures one step further. Instead of simply displaying the management procedure to follow, they could make operators run it via their Web browser, in case the procedure could be automated. This was typically achieved by using an HTML form: a proposed action would be displayed to the operator, and by clicking on a push button, the operator would trigger the execution of the adequate script—with no chance of making a mistake while typing in the commands in a `telnet` session. This possibility to interactively launch the execution of a troubleshooting procedure was a useful complement to symptom-driven HTML pages. Web

1. Note that some sites solved this problem before, on an *ad hoc* basis, by modifying the source code of the `telnet` client. But there was no general solution to this problem, and many NOCs relied on operators to `telnet` into a device and type in some commands in interactive mode.

technologies enabled operators to perform online troubleshooting, as opposed to simply getting online assistance.

5.2.2 Configuration management

Browser-based management also allows for a simple form of configuration management that leverages the ability of some agents to download their configuration file from a remote server upon reboot; e.g., Cisco IP routers use the Trivial File Transfer Protocol (TFTP) for that purpose. By interacting with HTML forms specific to the agent, the administrator generates a new configuration file on the Web server where the CGI program is executed. Then, this file is moved to the server where all of the agents' configuration files are stored. Finally, when the agent is rebooted by the operator, either manually or via the Web browser, it automatically loads its new configuration file. The advantage of this solution is that it works with legacy SNMP agents: Web technologies are only used to generate new configuration files. In Section 5.4.2, we will describe a more elaborate type of configuration management.

5.2.3 Java applet with an SNMP stack

Another type of browser-based management appeared several years later, with Java. In 1997–98, we saw a number of start-ups (AdventNet, Metrix, etc.) come up with new management applications developed partially or entirely in Java. In this scenario, the new, Java-based management application is independent of the traditional SNMP-based management platform. It complements it and runs in parallel. The Java-based management application provides operators with more user-friendly GUIs than HTML forms, which are somewhat limited in terms of graphics. The interaction with the agent still relies on well-known technologies, typically SNMP, or possibly others (e.g., `telnet` wrapped into `expect` scripts, or `ping`); but the front-end that operators interact with is now highly graphical. The management application can be implemented as a Java applet running in the operator's browser, or a standalone Java application—the difference in the code is limited. In the case of a Java application, the expression *browser-based management* is stretching it a bit, but the principle remains exactly the same.

This type of browser-based management is well suited to the sites who need simple online performance monitoring. Via a Web browser, it is possible to configure an SNMP agent (e.g., an intelligent hub supporting RMON) via a Java applet, then retrieve management data via SNMP for several minutes, process it within the applet, and finally display it with elaborate graphics.

The commercial market of Web-based management has thrived on this concept, with a slight nuance: the Java-based management application did not run parallel to, but instead of, the SNMP-based management application. Most of the start-ups in this market jumped on the bandwagon because it allowed them to take a share of a market that was until recently dominated by the near monopoly of the four main SNMP-based management-platform vendors. Note that, over time, several freely available software packages have also become available, including MRTG (Multi-Router Traffic Grapher [156]) and WebtrafMon [101].

5.3 Three-Tier Management

The last step before using HTTP end-to-end between the manager and the agent is to go via an external management gateway, typically an HTTP-SNMP gateway. We call it *three-tier management* because it relies on a three-tier management architecture. In this scenario, the communication between the Web browser and the gateway can be tightly or loosely coupled with the communication between the gateway and the agent. When it is tightly coupled, each request from the Web browser triggers a request from the gateway to the agent. When it is loosely coupled, the gateway independently retrieves and processes data from the agents, and requests from the Web browser are directly served from this aggregated data.

5.3.1 Deri *et al.*: SNMP-to-URL mapping

In 1996–97, the IBM Zurich Research Laboratory ran a project called Webbin'. This project leveraged on Web technologies to hide the idiosyncrasies of the SNMP and CMIP communication protocols to the management-application developer [18]. The core element of this project is the Liaison architecture. One facet of this work is a precursor of HTTP-based management: Deri's mapping [59, 60] between SNMP and Uniform Resource Locators (URLs [22]). In Liaison, an external management gateway[1] performs the translation between HTTP and SNMP. For instance, to retrieve the MIB variable `sysDescr` (description of the agent) in MIB-II, a manager can request the following URL from the management gateway:

```
http://kis.zurich.ibm.com/SNMP/GET/sysDescr.0?Host=bal.zurich.com&Community=public
```

In this example, `kis.zurich.ibm.com` is the management gateway and `bal.zurich.com` is the agent. A CGI program called `GET` processes the input on the gateway. It receives the OID `sysDescr.0` in input with two attributes: `Host` (the agent) and `Community` (the community string in SNMPv1 and v2c). In this scenario, the communication between the three tiers is tightly coupled: the manager sends an HTTP request to the gateway, which translates it into SNMP, sends an SNMP request to the agent, receives an SNMP response from the agent, translates it into HTTP, and sends it to the manager.

5.3.2 Kasteleijn: HTTP and SNMP

A second type of three-tier management was described by Kasteleijn in 1997 [117]. At that time, a new high-speed ATM backbone called SURFnet4 was deployed in The Netherlands. Kasteleijn developed a prototype called the Web-based ATM-Switch Management tool (WbASM) to allow institutions connected to the backbone to have access to management information related to it. SNMP data was prefetched by the WbASM server via SNMP, and stored in files. Users could later access the WbASM server from their Web browser and, after having authenticated themselves, access various usage statistics (e.g., link utilization) and instantaneous measurements (e.g., uptime per link). In this scenario, communication between the browsers and the WbASM server relies on HTTP, while communication between the WbASM server and the agents relies on SNMP. These two communications are loosely coupled.

5.3.3 Integration of a Web browser in the SNMP-based management platform

Another form of three-tier management is the integration of the Web browser into the SNMP-based management platform. Instead of using proprietary graphical interfaces, management-platform vendors can recode their GUIs as Java applets. For instance, by using the Java Native Interface (JNI), they can easily interface the Java servlets, which interact with the applets for the graphical part of the management application, with the rest of the application, which does the real work and is usually written in another language (e.g., C or C++) for a question of efficiency. In this scenario, operators use their Web browser as a single interface to all management tasks, while the SNMP-based management platform becomes an external HTTP-SNMP gateway that can operate in loosely and tightly coupled modes.

5.3.4 Deri: Java RMI and SNMP

In 1998, Deri described a three-tier approach to locate mobile equipment for asset management: JLocator [61]. This is the most sophisticated of all the approaches considered here in three-tier management, because it uses Java RMI to communicate between the Java applets (JLocator clients) loaded in the Web browser and the external management gateway (JLocator server). In other words, the JLocator server is an external RMI-SNMP gateway, as opposed to the external HTTP-SNMP gateways presented in the previous examples. For instance, once he/she is authenticated, the administrator can access detailed information about a roaming PC from the

1. In Liaison, this is called a *proxy*. But we use the term *management gateway* to comply with the terminology defined in Section 2.1.5.

Web browser: the Java applet displays information retrieved by the JLocator server from a database accessed via Java DataBase Connectivity (JDBC). Different JLocator servers can communicate via Java RMI, which allows management to be distributed. As in all the approaches presented in this section, the external management gateway communicates via SNMP with the agents. Note that JLocator is not simply a prototype: is has been deployed in production networks by Finsiel, Italy to locate thousands of assets in different sites.

5.4 HTTP-Based Management

The agents that we have considered so far do not support any Web technology. A milestone in Web-based management was the advent of embedded HTTP servers in network devices and systems. This led to *HTTP-based management*, whereby the manager and the agent communicate via HTTP. This type comprises two small variants. The first is depicted in Fig. 8. The user (administrator or operator) manages the agent via a management GUI running in a Web browser. This GUI is coded as a Java applet. On the agent, we can have different kinds of internal HTTP-SNMP gateways, as we will see in this section. The *SNMP MIBs* icon[1] represents all the MIBs supported by the agent, be they generic or vendor specific.

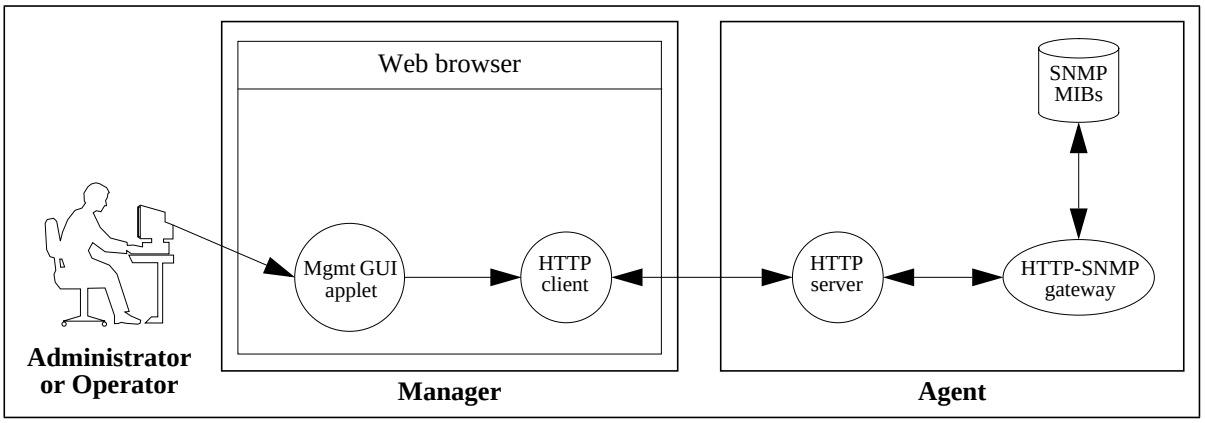

Fig. 8. HTTP-based management via a Java applet

The second variant is illustrated in Fig. 9. The agent side remains unchanged, but the management GUI is coded as a Java application on the manager side: the user does not use a Web browser.

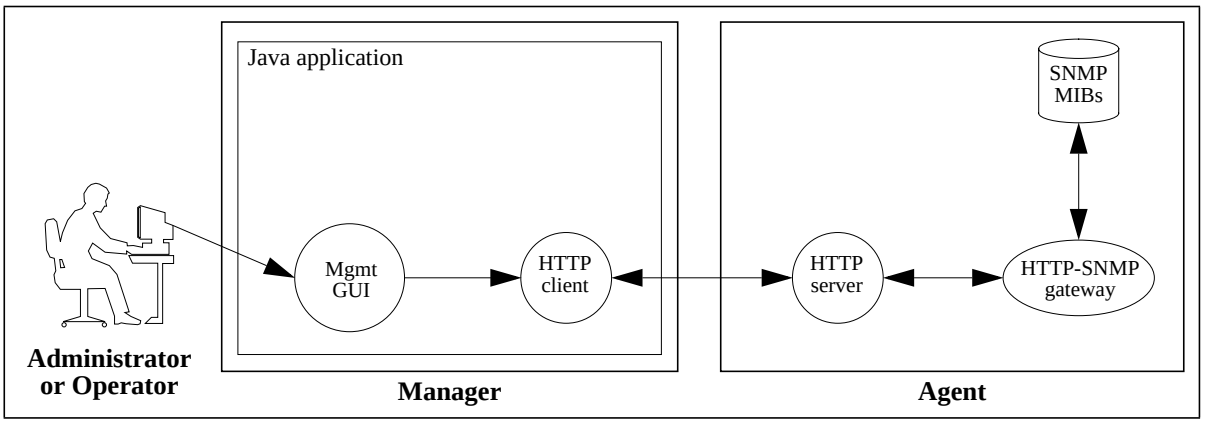

Fig. 9. HTTP-based management via a Java application

1. This icon is graphically represented as if the MIBs were stored on local storage, because a MIB is a virtual management-data repository. In practice, this data is not stored persistently (e.g., on disk or in EPROM) because that would be grossly inefficient, but it stays in volatile memory, generally as C data structures. An SNMP MIB is often implemented by agents as a collection of C pointers to proprietary data structures. Note that depending on the degree of optimization of the code run by the agent, the HTTP-SNMP gateway can directly access the MIB data structures in memory or do an explicit SNMP `get` or `set`.

Note that HTTP-based management is not necessarily interactive, unlike browser-based management for instance. The Java applet or application running on the manager side can perfectly communicate with the agent independently of any user action. As far as the agent is concerned, it makes no difference whether the manager side is an applet or an application, and whether it is running in attended or unattended mode.

HTTP-based management is still rarely used to date, except for configuration management. But despite its novelty, it is now a realistic option in NSM. In November 1998, at the Networld+Interop trade show in Paris, France, the author was told by three leading vendors in the IP networking market (Cisco, Nortel Networks, and 3Com) that they were routinely embedding HTTP servers in all their devices, except bottom-of-the-range ones. Most network devices sold today support HTTP, and most systems can easily support HTTP (many HTTP servers are freely available on the Web). If we consider that the average lifetime of a network device or system is about four years, we can state that a large proportion of the deployed equipment supports HTTP today.

5.4.1 CLI-to-URL mapping

In Section 5.2.1, we described the usefulness of `expect` scripts for wrapping interactive commands and running them in unattended mode. By emulating a `telnet` session, we have access to the entire CLI of a piece of equipment (unlike SNMP MIBs, for instance). This approach is not as flexible as it may first appear, though. The `expect` toolkit is good for executing routinely the same group of commands, but it is not appropriate as a general-purpose gateway to the agent's entire CLI. For instance, you do not want to create one `telnet` session per command line, because of the overhead incurred by the agent and the slowness of the process. Writing one `expect` script per command supported by the CLI does not scale. Another, more serious problem is the exposure to unexpected side-effects. To avoid the creation of one `telnet` session per command line, one could envision putting several command lines in a textual file and sending them in input to a general-purpose `expect` script. The problem with this approach is that the `expect` script needs to know exactly what the output looks like, in order to parse it and act accordingly. But the output of a command does not always depend on this command alone: sometimes it also depends on the sequence of commands that preceded it. Under these circumstances, if you allow for a random sequence of commands, you cannot guarantee that you know exactly what the output of the commands looks like. In real life, this is a no-no. You do not want to halt the operation of an agent by sending it a new, untested sequence of commands whose output does not correspond to what you expected. In production environments, `expect` scripts are normally confined to doing repetitive tasks, for which they are well debugged.

Embedded HTTP servers solve this problem: they make it possible to do via HTTP whatever is supported by the CLI—and especially whatever cannot be achieved via SNMP. Bruins [35] reports an experiment with CLI-to-URL mappings conducted by Cisco in 1995. For instance, a manager requests the following URL:

```
<http://router_name/exec/show/interface/ethernet0/>
```

The destination router is identified by its fully qualified domain name or IP address `router_name`. It runs an HTTP-SNMP gateway that translates this URL into the corresponding standard CLI command:

```
show interface ethernet0
```

The gateway forwards this command to the command-line interpreter, which interprets it as if it had been typed in interactively. The output of the command is sent to the gateway in return. The gateway translates it into an HTML page and sends it to the manager. The user can thus use his/her Web browser or Java application as if it were the console of the managed device or system.

This scheme presents two advantages. It is very simple to implement for vendors, because of the straightforward mapping between the CLI and URLs. It is also very easy to use for operators, because they are usually somewhat familiar with the CLI of the different vendors and do not have to learn yet another language. CLI-to-URL mappings therefore open new doors for configuration management and symptom-driven HTML forms, as there is no need to `telnet` into network devices or systems anymore. The operators who know a CLI by heart can also create URLs interactively, without having to write an `expect` script.

5.4.2 Embedded HTML pages and CGI programs

With embedded HTTP servers soon appeared embedded HTML pages, that is, static HTML pages that are locally stored on the agent, typically in EPROM. Some of these pages can be read-only, for instance to describe the characteristics of the system or network device. But the full potential of static HTML pages is unleashed when they are coded as HTML forms and coupled with embedded CGI programs. In this case, they are particularly useful for configuration management, because they enable administrators to set up network devices and systems in a more user-friendly way than what we saw in Section 5.2. By directly interacting with the agent via HTTP, administrators no longer have the inconvenience of first generating an external configuration file, then moving it to the right server, and finally loading it into the agent: they directly configure the agent.

Embedded CGI programs do not only work with static HTML pages: they can also generate dynamic HTML pages, on demand. This feature is particularly useful for performing simple performance monitoring, by updating a GIF image and refreshing the Web browser at regular time intervals[1]. This approach coincides with the rationale behind Management by Delegation, which promotes the delegation of a part of the management application to the agent itself. Clearly, a trade-off must be made between the agent's CPU cycles dedicated to management and the cycles devoted to the agent's operational task.

Static and dynamic embedded HTML pages, as well as embedded CGI programs, were all described in 1996 by Mullaney when he reported on a prototype developed by Cabletron [150]. Similar work at Cisco was reported in 1997 by Maston [137]. Over time, these features have gained a growing popularity in real life, especially for configuration management. A growing number of equipment vendors are now supporting embedded HTML pages and CGI programs. We expect that, in the near future, it will be the common way of configuring agents in interactive mode. Of course, interactive configuration is not always an option, especially in large networks. (We will solve this problem in our own management architecture.)

5.4.3 Embedded management application

The concept of *embedded management application* was described (and criticized!) by Wellens and Auerbach in 1996 [242]. Because agents are typically managed via vendor-specific management GUIs (add-ons) integrated in a management platform, these two authors argue that this add-on could be stored and run inside the agent itself. This is basically a variant of Goldszmidt's Management by Delegation, with different technical implementation details. The embedded management application is coded as a collection of HTML pages and CGI programs, and the operator directly interacts with it from his/her Web browser.

As with the previous approaches, this one uses HTTP rather than SNMP to vehicle data between managers and agents. The main advantage is that the network device or system is directly sold with its management application. The main drawback is that a person must sit in front of a Web browser to manage the agent: management cannot easily be automated. Another problem, not mentioned by Wellens and Auerbach, is the cost of management. With one add-on, you can manage an infinite number of devices of the same type from the same SNMP-based management platform. But if you now have to buy one embedded management application per managed device, the cost of management skyrockets when you manage many devices. To avoid this, you need to define a new business model for management software. A third problem, still not mentioned by the authors, but inherent in the concept of delegation, is that many CPU cycles are burnt on the agent for mundane tasks such as generating graphics and formatting the output for the user. The primary task of an agent is to fulfill its operational role, not to perform management tasks; so the amount of management overhead must be limited. In this scenario, with just a few clicks, an operator can unwillingly clog up an agent with management overhead, and thus temporarily hinder the execution of its operational tasks.

1. Netscape's Web browser supports a proprietary extension for refreshes: <META HTTP-EQUIV="Refresh" CONTENT=XXX> where XXX is the number of seconds that the browser must wait between two successive retrievals of the same URL [153].

5.4.4 Minimize the footprint of the embedded HTTP server

Two research teams have focused on the minimization of the embedded HTTP server's footprint on the agent: Reed *et al.* at IBM Almaden Research Center and Hong *et al.* at Postech, Korea.

In 1996–97, Reed *et al.* developed a minimal HTTP server that fits into a single C++ class. To achieve this, they had to remove several features of HTTP. HTML forms are not processed by external CGI programs but by the minimal HTTP server itself, which is multithreaded. Incoming HTTP requests specify what handler should process them (a handler is an HTTP server with a specific HTML-form parser). RPCs are used to perform the systems-management tasks requested via HTTP. This minimal server was used in NetFinity, a distributed management tool for PCs. The authors benchmarked their minimal HTTP server and showed that it uses less CPU than a conventional HTTP server that spawns a CGI program for each incoming HTTP request.

In 1999–2000, Hong *et al.* developed the POStech Embedded Web Server (POS-EWS) [56]. Its design goals were to minimize the overhead of HTTP-based management on the agent and to maximize the efficiency of the HTTP server. POS-EWS is a very compact piece of C code (30 kbytes). Its average runtime memory footprint is only 64 kbytes, which is remarkably low. Instead of being multithreaded as most HTTP servers freely available today, it is singlethreaded and runs multiple finite state machines to handle concurrent connections (see Welsh [243] for a description of the advantages of event-driven servers over threaded servers). POS-EWS was integrated in a commercial IP router developed by Hana Systems, Korea.

5.5 XML-Based Management

The next degree of sophistication in Web-based management is characterized by the use of XML to represent management data. We call this approach *XML-based management*. The most famous example is WBEM, which is backed by an important industrial consortium and currently has a lot of momentum. We will also describe a hybrid solution devised by John *et al.*, which uses both SNMP and XML for manager-agent communication. In Chapter 6, we will see that our own proposal (WIMA) can also be classified as a type of XML-based management.

5.5.1 Web-Based Enterprise Management (WBEM)

So far, WBEM's history has been exactly the opposite of SNMP's: after initially receiving a cold welcome from the research community, it is now raising high hopes. We explain this bizarre situation by reviewing a bit of history. We then summarize the main technical aspects that have already been specified by the DMTF.

A bit of history

The Web-Based Enterprise Management (WBEM) initiative was launched by an industrial consortium led by Microsoft in 1996 [26]. Its precise technical objective was blurred by sensational marketing announcements that it would integrate management by changing all existing management protocols and architectures. WBEM initially consisted of high-level definitions of a new information model called the HyperMedia Management Schema (HMMS), a new communication model limited to a new communication protocol called HyperMedia Management Protocol (HMMP), and a sketch of a new organizational model whereby clients access HyperMedia Object Managers (HMOMs) [100]. All deployed systems and devices worldwide were considered legacy systems that should be managed via management gateways, necessarily slow and awkward to use. By adopting this revolutionary approach and pretending to render obsolete all deployed management solutions, WBEM did not gain much credibility. Clearly, this was an instance of the *Reinvent the Wheel* antipattern [34].

Over time, the WBEM Consortium (that is, the vendors backing and specifying WBEM) became more realistic. In the communication model, new, domain-specific solutions were replaced with existing, standard ones. HMMP, which was supposed to be layered on top of HTTP, was dropped in favor of the well-known and pervasive HTTP; and the eXtensible Markup Language (XML) was selected to represent management data inside HTTP messages [27]. More importantly, the WBEM Consortium delivered at last: the information model, once known as HMMS, was renamed Common Information Model (CIM), and CIM 1.0 was officially released in April 1997 [226]. CIM 2.0, which is close to what we call CIM today, was released in March 1998. Finally, in the organizational model, HMOM became CIMOM (Common Information Model Object Manager).

In the summer of 1998, the responsibility for specifying the WBEM management architecture was formally handed over to the Distributed Management Task Force (DMTF), who integrated it into a global work plan [37]. This gave WBEM a much-awaited guarantee of independence *vis-à-vis* any particular vendor's interests. Since then, a lot of progress has been made, and the focus has clearly shifted from plain marketing to real technical work. WBEM, once considered vaporware, has now grown into a large-scale standardization process involving virtually all the major vendors on the market, especially in the areas of networking and systems. At the time of writing, the DMTF's work on WBEM is split across 17 Working Groups, all working on management issues ranging from applications to events, from databases to networks, and from security to SLA. In the entire 1990s decade, we have never had such a flavor of integrated management in the IETF realm. The DMTF has also learned from the IETF's mistakes, and is not only working on instrumentation MIBs, but also on high-level MIBs. As a result, a number of enterprises have moved the bulk of their NSM engineering forces from the IETF-backed SNMP management architecture to the DMTF-backed WBEM management architecture.

Let us now go through WBEM's main contributions to date: CIM (metaschema, schemata, and operations), CIM-to-XML mapping (xmlCIM), the CIM operations over HTTP, and DEN.

CIM metaschema

In DMTF parlance, a schema is (i) a collection of class definitions that describe managed objects and (ii) a namespace for these classes. This is what we are used to calling a MIB in NSM, or a model in information modeling (OMG). Similarly, a metaschema corresponds to a metamodel in information modeling or NSM. The terms *schema* and *metaschema* come from the database world, from which Thompson, the father of CIM 1.0, originates. The terms *model* and *metamodel* are well established in the OMG community. The IETF, ITU-T, and ISO are familiar with the term *model*. This change in the terminology has created some confusion in the NSM world, which is traditionally tied to the OMG world rather than the database world, especially due to CORBA. Note that to add to the confusion, a model for the DMTF is a large set of schemata, whereas a small set of schemata is still called a schema [38].

The main characteristic (and strength) of CIM is that it is object oriented. The CIM metaschema is specified in a document known as the *CIM Specification* [65]. The metaschema (metamodel) defines the entities that can be used to define CIM schemata (models): classes, properties (state), methods (behavior), qualifiers (metadata), etc. Version 2.2 of the CIM Specification was released in June 1999 and is now stable. Apart from this document, all WBEM-related specifications are regularly updated and should be considered ongoing work.

CIM schemata

CIM schemata are organized into three groups: the core schema, the common schema, and the extension schemata. Classes defined in the common schema are subclassed from those defined in the core schema. Classes defined in the extension schemata are subclassed from those defined in the core and common schemata. Finally, a management application can subclass from any of these schemata.

The *core schema* captures information that pertains to all areas of management. It contains high-level classes such as ManagedElement, ManagedSystemElement, PhysicalElement, LogicalElement, Service, System, etc.

The *common schema* is a set of schemata that are produced by different DMTF Working Groups. Its latest version (2.3) consists of eight schemata: User, Application, System, Device, Physical, Network, DAP (Distributed Application Performance), and Support. New ones are expected to appear in 2.4. The division between these schemata roughly corresponds to the different management areas that we consider in integrated management: networks, systems, applications, services, policies, etc.[1] The common schema therefore defines classes that are specific to a management area. For instance, the System schema defines classes such as UnitaryComputerSystem, SystemPartition, Cluster, ClusteringService, etc.

Finally, the *extension schemata* are specific to a given technology or vendor and are not directly controlled by the DMTF.

New versions of the core and common schemata are released jointly by the DMTF as a set of documents collectively called the *CIM Schema*. Version 2.3 was released in February 2000. Version 2.4 is currently under final review and is expected to come out shortly.

If we compare CIM with the SNMP information model, vendor-specific MIBs roughly map onto extension schemata, while generic MIBs map onto common or extension schemata. There is no equivalent to the core schema in the SNMP world.

CIM operations

CIM supports two types of methods (operations). *Extrinsic methods* are standard methods defined in a schema. *Intrinsic methods* are special methods directly invoked on a namespace. They provide a means to discover how another class is formed, and possibly modify remote classes or objects (instances of classes). This process is known as *reflection* in object-oriented software engineering. Some of these methods are read-only, e.g., `EnumerateClasses`, `EnumerateInstances`, `GetProperty`, and `GetQualifier`. Others allow for modifications, e.g., `DeleteClass`, `DeleteInstance`, `CreateClass`, `CreateInstance`, `ModifyClass`, `ModifyInstance`, `SetProperty`, etc. Because all namespaces are instances of the class `__NameSpace`, or instances of a class subclassed from it, and because all CIM servers must support the `root` namespace, all standard operations on classes and instances also apply to namespaces. For historical reasons, CIM intrinsic methods are specified in Section 2.4 of the specification for CIM operations over HTTP [68], although they are totally independent of HTTP.

CIM-to-XML mapping (xmlCIM)

The representation of CIM in XML is known as *xmlCIM* in the DMTF realm. As this name is a bit bizarre, we call it *CIM-to-XML mapping* in this work, because we find it more self-explanatory. The latest specification is version 2.0, released in July 1999 as two separate documents: an XML Document Type Definition (DTD) [66] and a clear text explaining the conventions cast in iron in the DTD [67]. The CIM-to-XML mapping defines XML elements, entities, and attribute lists to represent the information stored in CIM schemata. For instance, these documents specify how to represent a class, a method, a property, etc. We will come back to the CIM-to-XML mapping in detail in Section 8.2. In particular, we will explain that the DMTF opted for a metamodel-level mapping.

CIM operations over HTTP

The specification for CIM operations over HTTP complements the previous document. Version 1.0 was released in August 1999 [68]. Unlike the previous specification, which is independent of the communication protocol used to transfer XML documents, this one is specific to HTTP.

The encapsulation of CIM operations in HTTP uses a number of standard HTTP header fields, and defines several error codes that are compliant with the HTTP/1.1 specification [74]. It also uses HTTP extension

1. One aspect that we did not take into account is user administration, covered by the User schema.

headers [155], an add-on to HTTP/1.1 that allows HTTP clients and servers to add new, application-specific HTTP header fields to the standard HTTP/1.1 header fields. An HTTP server is warned that an incoming request uses extensions by the use of the prefix "M-" in front of the HTTP method. CIM operations over HTTP use the `M-POST` HTTP method. The prefix "M-" means that this HTTP `POST` request contains at least one mandatory extension header field. Upon receipt of such a request, and prior to processing it, the HTTP server must tell the HTTP client whether it understands all the extensions used in the HTTP header. Six HTTP extension headers have been defined by the DMTF: `CIMOperation`, `CIMProtocolVersion`, `CIMMethod`, `CIMObject`, `CIMBatch`, and `CIMError`. Note that HTTP extensions are currently published as an experimental RFC, and are far from being supported ubiquitously.

DEN

Directory-Enabled Networks (DEN) is a simple concept popularized by Strassner [214], initially promoted by Cisco, and now backed by many vendors under the umbrella of the DMTF. DEN consists of a data repository and a naming service. The data repository is called a *directory*, but we call it a *DEN directory* to avoid any confusion with Unix directories, Windows directories, etc. DEN has a strong flavor of object-oriented modeling, although Strassner insists that DEN is not object oriented. The naming conventions and the concept of a *directory tree* are simplified versions of X.500 [109]. Information is stored hierarchically. Each entry (object) is stored as a set of attributes. DEN directories distribute management information across multiple machines through a process known as *directory replication*. There are built-in mechanisms for ensuring replication and synchronization between different machines. DEN directories are typically accessed via the Lightweight Directory Access Protocol (LDAP), which supports three types of protocol operations: query, update, and authentication. The schema of the DEN directory is defined as the complete set of classes and attributes for that directory. Version 1.0 of the guidelines for CIM-to-LDAP directory mapping was released in May 2000.

5.5.2 John *et al.*: XNAMI

Another example of XML-based management was described by John *et al.* in 1999 [114]. In the XNAMI architecture, the agent maintains an explicit runtime representation of its MIBs (unlike most SNMP agents) and uses XML to represent managed objects internally. For each OID, the agent stores Java bytecode for the `GET` and `SET` methods. The agent's SNMP server is modified so as to execute the `GET` method of an OID upon receipt of an SNMPv3 `GET` (ditto with `SET`). This feature allows the agent to support dynamically extensible MIBs, as opposed to standard, statically defined SNMP MIBs. OIDs can be added or removed at runtime by using the Document Object Model (DOM [232]) API. The agent supports a new MIB called the XNAMI MIB, which defines two new OIDs: `methods_proxy` and `mib_proxy`. By sending an SNMP `SET` PDU containing the OID `methods_proxy`, the XNAMI manager can transfer compressed Java bytecode for the `GET` and `SET` methods of any OID (mobile code paradigm). By sending an SNMP `SET` PDU containing the OID `mib_proxy`, the XNAMI manager can transfer an XML string specifying that a new OID should be created, an existing OID deleted, etc. The compressed Java bytecode and the XML string are both transferred as BER-encoded SMIv2 strings. The XNAMI manager is implemented as a Java servlet, the XNAMI agent as a Java application. The XNAMI manager and agent exchange management data via SNMPv3, while the Web browser and XNAMI manager communicate via HTTP. The two communications are tightly coupled. We call this approach hybrid because XML data is transferred via SNMPv3 rather than HTTP.

5.6 Distributed Java-Based Management

The highest degree of sophistication in Web-based management is what we call *distributed Java-based management*. It is based on Java RMI for distribution aspects and is a variant of the distributed-objects approach described in Section 3.1.6.2. We already ruled out this option for the next management cycle in Chapter 4, because of the *my-middleware-is-better-than-yours* syndrome. But it is a valid form of Web-based management, so we summarize it here for the sake of completeness. The main industrial supporters of this technology are Sun Microsystems and the Java Community; so far, they have proposed three solutions: JMAPI, JMX, and FMA. Anerousis has also developed an elegant solution based on Java RMI: Marvel.

5.6.1 Java Management Application Programming Interface (JMAPI)

Java RMI was officially released in early 1997 with JDK (Java Development Kit) 1.1. At the end of 1996, when RMI was still a beta release, Sun Microsystems leveraged it to specify a radically new way to manage networks and systems: the Java Management API (JMAPI [221]). This API is a set of tools and guidelines to build management applets supporting Java RMI. In this approach, everything is an object: every managed object in every SNMP MIB is mapped onto a full-blown Java object, and the manager can interact directly with the JMAPI managed objects in the agent. On the manager side, a JMAPI object acts as an SNMP trap handler, receiving all incoming SNMP traps from non-JMAPI agents and converting them into JMAPI events. The Notification interface allows administrators to develop event-driven management applications. In the evaluation code that Sun Microsystems made freely available on the Web in 1997, MIB-II was entirely translated into JMAPI managed objects.

Strangely enough, one of the main reasons of JMAPI's success eventually became the main cause of its dismissal. In those days, the JDK was still lacking a proper library of graphical components. Swing did not exist, and programmers had to interact directly with the low-level AWT (Abstract Window Toolkit). JMAPI came with its own library of high-level graphical components, offering most of the widgets then available in Motif, and many people used it to build all sorts of applications, not necessarily related to management. Later, once Sun Microsystems had released JavaBeans and Swing, JMAPI was swiftly abandoned. But by then, it had already demonstrated the possibility of viewing management as a distributed object-oriented application.

5.6.2 Java Management eXtensions (JMX)

In the Java world, Jini [216] promotes a vision similar to Microsoft's Universal Plug and Play in the Windows PC world: you buy a device, you connect it to your network; by itself, the device finds its IP address, registers with a naming service, describes what functionality it supports, gets its configuration data, etc. A real administrator's dream! Unfortunately, Jini's vision of management is simplistic and limited to automated registration and configuration in a DEN-directory-like repository. JMX [218] filled this void with a comprehensive management framework, in the object-oriented sense of the term.

Once known as JMAPI 2.0, JMX was officially released to the public in May 1999, at the JavaOne conference. It is a form of object-oriented and component-oriented Web-based management. Most of the work so far has concentrated on the agent side [219]. JMX does not define a new information model: it interfaces with existing ones. Two APIs have been specified for SNMP and WBEM/CIM, and a TMN API is currently under definition. The agent and the manager can communicate via Java RMI, HTTP, or SNMP. As far as SNMP is concerned, JMX relies on a general-purpose SNMP-to-Java MIB compiler that translates the managed objects defined in any SNMP MIB into components called *MBeans* (short for *Management Beans*). It is not yet clear what technology should underpin the manager side, possibly EJBs. The latest JMX specification leaves its definition for a future phase [219, p. 21].

5.6.3 Federated Management Architecture (FMA)

Independent of JMX, the Federated Management Architecture (FMA [220]) was devised in 1999 by the industrial consortium in charge of developing Jiro, a Java technology for storage management. Initially begun as an effort to standardize management in the storage industry, FMA grew into a full-fledged, general-purpose management architecture. The initial concept of *FederatedBeans* was dropped in favor of standard JavaBeans and EJBs, thereby addressing the main problem with Jiro. With FMA, all agents must be Jini-enabled. FMA defines static management services (e.g., the event service, the log service, and the controller service), and dynamic services that extend Java RMI semantics to the management-application level. FMA also specifies management aspects related to security, transactions, persistence, etc. Manager-agent communication can rely on any protocol: SNMP, HTTP for WBEM, etc. The manager side of FMA is completely specified. Some work is under way to unify JMX and FMA into a single management architecture for Java-based management.

5.6.4 Anerousis: Marvel

Work on distributed Java-based management has not been confined to Sun Microsystems and the Java Community. In the research community, we have also seen several proposals and prototypes that use Java RMI. The most sophisticated and comprehensive solution published to date is Marvel by Anerousis [7, 9]. Marvel is a distributed, object-oriented management architecture, as well as an NSM platform. At the architectural level, it supports Java RMI, CORBA, SNMP, CMIP, and DMI for manager-agent communication; but in practice, object services are implemented with Java RMI. Marvel's information model supports the generation of computed views of management information [8], 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. This data is aggregated into Marvel objects (also known as *aggregated managed objects*) using spatial and temporal filters. Marvel objects can reside in the agents or in external repositories, depending on the amount of processing required (some aggregations can be very CPU intensive). This makes Marvel's management architecture very scalable. Note that this aggregation in external nodes applies to the IP world the TMN concepts of element-level MIBs and network-level MIBs: Marvel objects can serve as the building blocks to a full-fledged Network Information Model (NIM) in the IP world.

5.7 Commercial Products

The Web-based management market is currently very active, with many start-ups coming in this lucrative market every year. Publishing an up-to-date list of commercial products in this area is therefore doomed to failure. For instance, in 1999, Harler [92] published a book that includes a 141-pages list and analysis of commercial software packages in the area of Web-based management. Terplan [225], the same year, published a similar list of 79 pages. One year later, both of them are already obsolete. For instance, AdventNet, Metrix, Rapid Logic, and SNMP Research all offer Web-based management packages today, but none of them is referenced by Harler or Terplan.

A more practical approach is to publish such compilations on the Web, to allow for dynamic updates. For instance, during several years, Lindsay maintained a very useful list of commercial packages on Web-based management [128]. Unfortunately, this list does not appear to be updated anymore. The SimpleWeb is another Web site maintaining a list of commercial network-management software packages [192]. Note that it is not dedicated to Web-based management.

5.8 Summary

In this chapter, we have presented the state of the art in Web-based management. In Section 5.1, we began with browser-based metamanagement, an entry-level approach that deals with collateral management tasks. In Section 5.2, we described browser-based management, a simple approach whereby the agent supports no Web technologies and the manager directly accesses the agent via a Web browser. In Section 5.3, we detailed three-tier management, whereby the manager and the agent communicate via an external management gateway. The manager can use advanced techniques to communicate with the gateway, whereas the agent is still a plain SNMP agent. In Section 5.4, we reviewed HTTP-based management, a more sophisticated approach that leverages an embedded HTTP server in the agent to communicate via HTTP between the manager and the agent. In Section 5.5, we studied XML-based management, an advanced approach that is epitomized by WBEM. This time, the agent not only supports HTTP, but also XML. In Section 5.6, we completed our taxonomy of Web-based management with distributed Java-based management, whereby the manager and the agent communicate via an object-oriented middleware. Unfortunately, this very elaborate approach is exposed to the *my-middleware-is-better-than-yours* syndrome. Finally, we gave pointers to lists of commercial software packages in Web-based management.

This concludes the first part of our dissertation. Now that we have defined the problem at stake, investigated the possible solutions, selected Web-based management, and reviewed in detail all the current approaches in Web-based management, lets us now present our contributions in the next chapters.

Chapter 6

A NEW MANAGEMENT ARCHITECTURE: WIMA

In this chapter, we propose a new management architecture for IP networks and systems in replacement of the SNMP management architecture used today. Our architecture is called WIMA (Web-based Integrated Management Architecture) and consists of two parts: a push-based architecture for regular management and notification delivery, and a pull-based architecture for *ad hoc* management. Our main innovations lie in the former, which combines Web and push technologies to communicate between agents and managers (or between managers in distributed hierarchical management). We claim that WIMA solves the problems identified in SNMP-based management in Section 2.4, and meets the new requirements set by the market. We also propose an elegant way of dealing with legacy agents which, at the same time, paves the way for future agents supporting yet-to-be-defined information models.

Although we actually used both top-down and bottom-up approaches to devise WIMA, this chapter is organized in a top-down manner for the sake of clarity. In Section 6.1, we draw the main lines of our management architecture (analysis phase) and outline its novel organizational and communication models. In Section 6.2, we present our main design decisions (design phase) and point out the most significant differences between the SNMP and WIMA management architectures. In Section 6.3, we describe our push-based architecture (WIMA-push) and its organizational model; we also propose a migration path to gradually implement them in real-life networks. In Section 6.4, we introduce our pull-based architecture (WIMA-pull) and its organizational model, and again describe a migration path. Finally, we summarize this chapter in Section 6.5. The communication model of our management architecture will be presented in Chapter 7, with XML complements in Chapter 8.

6.1 Main Architectural Decisions (Analysis Phase)

In this section, we analyze the main architectural decisions behind our proposed management architecture, and the main differences between the SNMP and WIMA management architectures. We also explain why we focused our attention on the organizational model (WIMA-OM) and communication model (WIMA-CM).

6.1.1 One management architecture, four models

In NSM, management architectures are traditionally broken down into four models[1]: an information model, a communication model, an organizational model, and a functional model [98, 50]. Which of these models should be changed to solve the problems listed in Section 2.4?

New functional model?

For many years, there has been a great consensus around the management tasks defined by the functional model. These tasks are traditionally grouped into five functional areas: fault, configuration, accounting, performance, and security [48]. Even the OSI and SNMP research communities agree on this! As new techniques appear or are better understood, new functions are defined; recent examples include the Command Sequencer [113] in the OSI world and the Script MIB [126] in the IP world. But there is no fundamental questioning of the OSI/SNMP functional model itself. We therefore chose to keep it unchanged in WIMA.

Apart from the functional model, there is no consensus at all in the research community as to which model should be preserved and which should be changed to address SNMP issues.

New organization model?

Throughout the 1990s, we saw many proposals to radically change the organization model (see Chapter 3). The most relevant to us are the following:

- Management by Delegation [88] proposed a new organizational model, but it did not go as far as specifying a new communication model. Goldszmidt's prototype used a new communication protocol (the Remote Delegation Protocol), but it was a proof of concept to demonstrate the gains of his new organizational model rather than a plea to use a specific communication model.
- Along the same line, the Script MIB [126] proposed a change in the SNMP organizational model to delegate tasks to agents. It defined a new management task, thereby complementing the SNMP functional model, and a new MIB for storing related management data. But it kept the SNMP communication model unchanged.
- Many proposals based on mobile code were also made: some in the area of active networks, others in mobile agents, etc. (see Section 3.1.6.1). In NSM, we still have not seen any proposal mature enough to specify a new organizational model. The main problem with mobile code is security, and as long as this problem remains unsolved by software engineers, mobile-code technologies will remain confined to prototyping in NSM (see Chapter 4).
- Finally, the engineering problems in multi-agent systems, and DAI in general, are such that we are no where near seeing a detailed proposal for a new organizational model for NSM, let alone a new, comprehensive management architecture (see Chapter 4).

New information model?

During the same decade, we also saw two attempts to replace the SNMP information model:

- The OSI management metamodel, made up of GDMO [53] and the General Relationship Model (GRM [112]), is much richer than the SNMP metamodel (SMIv2 [44]). Said otherwise, the expressiveness and semantic richness of information modeling are much higher in OSI management. The OSI management metamodel is object-oriented and supports named relationships, actions, etc. [85]. All of these are missing in SNMP, so programmers and modelers resort to ugly workarounds such as programming by side-effect[2]. To address these deficiencies in SNMP, some people suggested using OSI information modeling in the IP world. This led to a "religious war" between the IP and telecom world [23].

1. Hegering *et al.* call them *submodels* [98].
2. You set an MIB variable to a certain integer value, which makes the agent invoke a certain action, e.g. "reboot".

Although the expressiveness of OSI information modeling was undoubtedly superior to SNMP's, the market did not adopt it because of the very design of OSI. The four models of the OSI management architecture are deeply intertwined; by snowball effect, the adoption of its information model requires the adoption of its communication model (CMIP and CMIS), its seven-layer OSI stack, its connection-oriented transport layer, etc. SNMP was created precisely to avoid this complexity, thus OSI information modeling never made it to the IP world.

- Several years after this "religious war" was over, the DMTF issued a new object-oriented information model for the IP world: CIM [38]. By being object-oriented, CIM addresses the main shortcoming of the SNMP metamodel, and increases the expressiveness and semantic richness of information modeling in NSM. Another advantage of using CIM is that it can be independent of the communication model: adopting CIM does not mandate the adoption of an entire protocol stack[1].

New communication model?

Despite well-known problems in the SNMP protocol (see Chapter 2), little attention has been paid to the communication model in the past decade. Only the DMTF has made a proposal [68]. After erring for some time with a new transfer protocol (HMMP), the DMTF decided that agent-manager transfers should rely on HTTP instead of SNMP. This proposal specifies the encapsulation of XML-encoded CIM operations in HTTP, and HTTP/1.1 extensions to facilitate the traversal of firewalls. But it leaves several aspects of the communication model unspecified.

New management architecture?

The DMTF actually claims to have defined a brand-new management architecture: WBEM. WBEM indeed includes a new information model. The CIM metamodel[2] is now fairly stable [65], and many CIM schemata are currently under development. But we just saw that its communication model is only partially specified. And it does not at all specify the nature of agent-manager and manager-manager interactions (organizational model). So, as it stands today, the WBEM management architecture consists of (i) an information model, (ii) a partially specified communication model, (iii) an implicit functional model, and (iv) no organizational model. As a result, Web-based management-platform vendors (mostly start-ups today) are encouraged to simply add support for a new information model to their existing SNMP-based platforms, and to tweak HTTP proxies and servers to support the DMTF's HTTP/1.1 extensions for going across firewalls, without changing anything to the way managers and agents interact. Therefore, the WBEM management architecture cannot be considered today as an alternative to the SNMP management architecture.

WIMA

In summary, many efforts have been devoted in the past decade to defining new organizational models, some efforts to defining new information models, and none to defining both a new organizational model and a new communication model. This might explain why none of them are widely adopted today, and why the SNMP management architecture is still the only reasonable alternative in the IP world.

We filled this void with WIMA, our new management architecture. We changed the way managers and agents interact and exchange management data, we studied how this change affects the four different models, and we specified new communication and organizational models. We also kept the information model orthogonal to the communication and organizational models, which allows us to deal with agents supporting any information model: SNMP MIBs, CIM schemata, etc. Let us investigate these different architectural decisions in more detail in Sections 6.1.2 through to 6.1.5.

---

1. We will see in Chapter 10 that the DMTF has now bound its communication model to its information model, so the two are not necessarily independent in WBEM. But they can be made independent, as we will demonstrate in Chapter 7 with WIMA-CM.
2. Or *metaschema* in DMTF parlance, see Section 5.5.1.

6.1.2 No need to define yet another information model

The integrated-management community has been plagued for many years by “religious wars”. One of them is about informations models. Whenever a new problem (or idea) comes up, some people believe that it should be solved (or implemented) by inventing a new *ad hoc* information model. For instance, we saw this problem already with network-topology information management [157]. Saperia and Schönwälder also exposed it in policy-based management [180]. This rationale “forgets” that the success encountered by the SNMP management architecture in the 1990s owes a great deal to two facts: we had a single information model to manage IP networks, and a very stable metamodel (only two variants in 10 years: SMIv1 and SMIv2). The time it takes to educate a large market with new information modeling techniques should not be underestimated, and the number of information models that need to be mastered by management-application designers, programmers, and administrators should be kept to a strict minimum.

The DMTF recently released a new information model: CIM. By replacing data-oriented SNMP MIBs with object-oriented CIM schemata, CIM addresses the main shortcoming of the SNMP information model. In view of the number of vendors now backing the DMTF, there is little doubt that CIM will get some share of the information-modeling market. Whether CIM schemata will eventually replace SNMP MIBs, or simply complement them, remains to be seen and will only partially be based on technical merits. But now that we have two information models at our disposal in the IP world (one, very simple and widely deployed, and another, conceptually rich), we believe that we have enough information models to cover all our needs for the next management cycle in NSM and beyond (that is, in integrated management at large). What are needed today are new SNMP MIBs or new CIM schemata, not a new metamodel, not a new way of modeling management information. In WIMA, we therefore did not specify yet another information model.

6.1.3 Dissociation of the communication and information models

We actually went one step further and decided that our management architecture should be orthogonal to the information model(s) supported by the agents and managers. In other words, WIMA does not prescribe the use of a specific information model but copes with any information model. With a single communication model, we can transfer management data pertaining to many different information models: SNMP MIBs or CIM schemata for IP networks and systems, OSI MITs for hybrid networks (when we mix IP and telecoms networks), etc. We made this possible by completely dissociating the information and communication models. Each bundle of management data sent by an agent to a manager is self-describing, with metadata indicating the type of management data being transferred (e.g., SMIv2-compliant MIB variable encoded in BER, or CIM 2.2 object encoded in XML). To make this possible, we dropped the SNMP communication protocol in favor of HTTP, and proposed an original way of structuring data in persistent HTTP/TCP connections. The details of our solution will be presented in Chapter 7.

With this architectural change, new information models can be defined over time, if need be, without altering the communication model. If CIM were replaced tomorrow with a new, more efficient object-oriented information model, we could still use the same communication model in our management architecture. When a new information model comes up, we only have to define a way to encode management data and encapsulate it in HTTP. This characteristic is very important to deal with legacy systems, especially already deployed SNMP MIBs (see Section 6.2.7). It is also important for the future because it does not mandate an information model that could prove to be too limiting in the future (as we experienced with SNMP, for instance).

Our architectural decision to completely dissociate the information and communication models represents a major breakaway from the SNMP and OSI management architectures. The close ties between these two models are, in our view, a conceptual mistake in the OSI and SNMP management architectures. The DMTF has not yet fully defined a communication model for its WBEM management architecture, but the definition of CIM-oriented HTTP extensions to facilitate the traversal of firewalls leads us to believe that the same mistake is about to be repeated (we will come back to this in Chapter 10).

6.1.4 A new organizational model: WIMA-OM

In our organizational model, we changed the way agents and managers interact and specified how management should be distributed. One of our main innovations is to use push technologies for transferring notifications *and* regular management data between agents and managers (or between managers in distributed hierarchical management), and not simply for delivering notifications as in SNMP-based management. Let us first summarize the main differences between the push and pull approaches. We will then present the advantages of push over pull to transfer regular management data, and justify why the pull model should be preserved for *ad hoc* management. Finally, we will describe how to distribute management.

6.1.4.1 Push model vs. pull model: two organizational models

In software engineering, the pull and push models designate two well-known approaches for exchanging data between distant entities. The newspaper metaphor is a simple illustration of these models. If you want to read your favorite newspaper everyday, you can either go and buy it every morning, or subscribe to it once and then receive it automatically at home. The former is an example of pull, the latter of push.

In NSM, the pull model is based on the manager-agent paradigm, a variant of a standard paradigm in software engineering: the request-response paradigm, typically used in client-server architectures. The client sends a request to the server, then the server answers, synchronously or asynchronously. In SNMP-based management, this is called *data polling*, or simply *polling*. It is functionally equivalent to the client "pulling" management data off the server. In this approach, the data transfer is always initiated by the client, i.e. the manager. This model is the basis for most management-data transfers in the SNMP and OSI management architectures.

The push model, conversely, is based on a variant of the publish-subscribe design pattern[1] [40]. In this model, the agent (or the mid-level manager in distributed hierarchical management) advertises what information model it supports (SNMP MIBs, CIM schemata, etc.), and what notifications it can send (SNMPv1 traps, SNMPv2 notifications, CIM events, etc.). Then, the administrator subscribes the manager (or the top-level manager in distributed hierarchical management) to the data he/she is interested in, specifies how often the manager should receive this data, and disconnects. Later on, each agent individually takes the initiative to push data to the manager, either on a regular basis via a scheduler (e.g., for network monitoring) or asynchronously (e.g., to send SNMP notifications). In the SNMP and OSI management architectures, only notification delivery follows the push model.

In WIMA, our organizational model uses both approaches. The push-based organizational model is called WIMA-OM-push; the pull-based organizational model is called WIMA-OM-pull.

6.1.4.2 WIMA-OM-push for regular management and notification delivery

For IP networks and systems, the pull model has been used for over a decade for regular management and *ad hoc* management, while the push model was used only for notification delivery. Yet we claim that the push model is better suited to regular management than the pull model. In WIMA, we use push technologies for both regular management (data collection and monitoring) and notification delivery. The advantages of using push technologies in NSM are fourfold: we save network bandwidth, we transfer some workload to the agents, we improve scalability, and we facilitate the support for redundant managers, thereby improving the robustness of the management application.

Save network bandwidth

Much of the network overhead caused by SNMP polling is due to the fact that data collection and monitoring are very repetitive: there is a lot of redundancy in what the manager keeps asking all the agents. For instance,

1. The publish-subscribe design pattern is called the *Observer* pattern by Gamma *et al.* [84].

in SNMP-based network monitoring, a common way to check if a machine is still up and running is to request the same MIB variable, typically its `sysObjectID` (MIB-II), every few minutes. This scheme works but is very inefficient, as the manager marshalls and sends the same OID to all the agents, at every polling cycle, endlessly. Instead, with the push model, the manager contacts each agent once, subscribes to this OID once, and specifies at what frequency the agent should send the value of this MIB variable. Afterward, there is no more traffic going from the manager to the agent (see Fig. 10). All subsequent traffic goes from the agent to the manager (except in the rare cases when the administrator wishes to update the list of OIDs that the manager is interested in, or change a push frequency). So, by using push technologies instead of polling, we reduce the network overhead of management data, thereby saving network bandwidth.

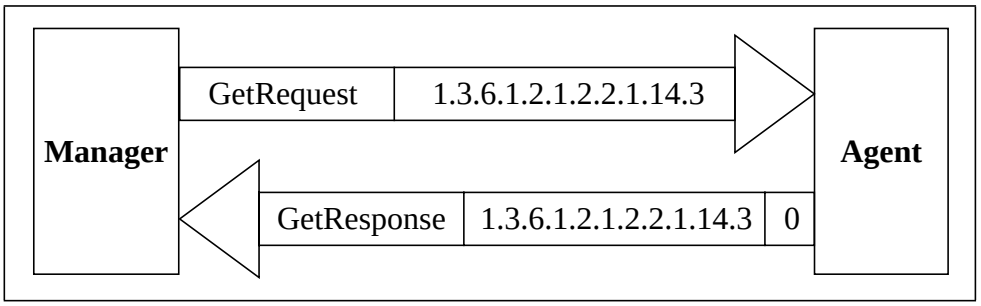

Fig. 10. Network overhead of an SNMP `get`

How much do we save? Is this difference significant or marginal? For instance, in SNMP, the error rate for inbound traffic through interface #3 is given by the OID `1.3.6.1.2.1.2.2.1.14.3` in MIB-II [143], as depicted in Fig. 10. The manager issues a `get` request to the agent. It consists of a small header specifying that this is a `GetRequest` PDU, and a long OID that indicates the MIB variable of interest to the manager. The agent sends a reply that consists of a small header (`GetReply` PDU), a long OID, and a short value for that MIB variable. So, with an SNMP `get`, the same OID is sent twice over the network, and accounts for most of the network overhead. With push technologies, the manager no longer sends a `get` request for each MIB variable for each agent. Once the subscription is performed, the network overhead is only due to the agents pushing {OID, value} pairs to the manager. As the value and header take little space compared to the OID, we roughly halve the traffic by going from pull to push technologies.

Under certain circumstances, the network-bandwidth saving can be even greater. In SNMP, MIB traversals or table retrievals are usually achieved with `get-next`. In this case, three OIDs are transmitted instead of two: in its reply, the agent not only includes the OID described above, but also the next OID that the manager should request. By using push technologies instead of pull, we then divide by about three the network overhead caused by transfers of management data.

In short, compared with SNMP polling, push technologies significantly reduce the network overhead caused by management data.

Transfer some of the workload to the agents

The second advantage of using push technologies is to delegate some of the processing from the manager to the agents (or from the top-level manager to the mid-level managers in distributed hierarchical management). In the SNMP management architectures, the agent is supposedly dumb, so the manager does everything. This makes it easy to cope with bottom-of-the-range, as well as top-of-the-range, network equipment, but it also causes significant CPU overhead on the manager. SNMP was really designed to cope with small LANs, where the aggregated overhead is easily bearable for the manager. With large networks, or with small networks where the manager has to execute a large number of rules (e.g., LANs experiencing instability), the total processing overhead can be unbearable for the manager, which can become a bottleneck and slow down the entire management application. One way to solve this problem is to transfer some of the CPU burden from the manager to the agents, as advocated by Goldszmidt with Management by Delegation [88], or Wellens and

Auerbach when they exposed the Myth of the Dumb Agent [242]. The main argument behind this transfer is that more and more agents have powerful microprocessors and significant memory, thereby rendering the vision of a “dumb” agent obsolete.

By transferring some of the workload to the agents, we not only free the manager from some of its CPU load, but we also decrease the requirements put on the manager in terms of CPU and memory. In large networks, management platforms are often big servers that cost a fortune to buy and maintain. Agents, on the other hand, are more powerful than they used to be, and most agents can reasonably do a bit of processing locally.

Improve scalability

The third advantage of push technologies is that they improve the scalability of NSM. This is the direct result of saving network bandwidth and transferring some of the workload to the agents. By reducing the network overhead per transferred management data (e.g., per SNMP MIB variable), we make it possible to transfer more management data, that is, to manage more agents from a single manager or to transfer more management data per agent. And by transferring some of the CPU workload to the agents, we free some resources on the manager, thereby allowing it to cope with more management tasks. As we expect the management-related traffic to increase in the future, we improve the scalability of NSM if we transfer data between agents and managers in a more efficient way.

Improve robustness through redundancy

A fourth advantage of the push model is that it makes it easy to support redundant managers, either through IP multicasting or duplicated push. Although most network devices and systems to date are managed by a single manager, some environments require two (or possibly more) managers to run in parallel to allow for automatic failover and thereby improve the robustness of the management application.

The way to achieve this is fairly simple with push technologies. When an administrator subscribes to some management data (see Section 6.3.3), he/she tells the agent what manager it should send the data to. Instead of specifying a single unicast IP address, the administrator can specify a multicast address or several unicast addresses. For an agent, sending data to a unicast or a multicast address is identical, and sending it to two or three unicast addresses is not immensely different. The only requirement in the first scenario is that it should support IP multicasting, which modern implementations of the TCP/IP stack generally do. Management data is thus sent to multiple managers in parallel, which makes the management application more robust. For instance, one manager can crash while another takes over transparently (“hot standby”). Alternatively, standby managers can receive data passively until they are configured to replace the master manager (“cold standby”).

With pull technologies, we can also have the agent send management data to several managers. But in this case, all of the managers have to request the data independently, which increases the network overhead significantly (we multiply it by the number of managers) and consumes more CPU cycles (we multiply the overall CPU overhead by the number of managers).

Even if this improved robustness is still a long way from fault-tolerance, it can be very attractive to organizations whose network is of critical importance to the smooth running of their business, but who cannot afford expensive, full-blown, fault-tolerant systems.

6.1.4.3 WIMA-OM-pull for *ad hoc* management

Push technologies beat polling in many respects, but there is a price to be paid: the overhead caused by the configuration of the agent during the subscription phase. Obviously, push is superior to pull if, and only if, this overhead is outweighed by the network-overhead gains described earlier. This is the case only when the same MIB variable is transferred many times from an agent to a manager.

For *ad hoc* management, by definition, we are in the opposite situation. We want to retrieve a MIB variable just once, or perhaps ten times in a row. In such cases, the overhead of configuring the agent is not offset by the gain of pushing management data from the agent to the manager. So we have a threshold effect. The value of this threshold can only be determined empirically. For troubleshooting, whenever we want to monitor a MIB variable over a short period of time (typically, up to 5 minutes), we are in the realm of *ad hoc* management and should use pull technologies; beyond that threshold, we are in the realm of regular management and should use push technologies. Note that there is an overlap zone where it does not really matter whether we use push or pull technologies, as the relative gains or losses are negligible.

6.1.4.4 Distributed hierarchical management

As we saw in Chapter 2, distribution was a late concern in SNMP. When the M2M MIB was rendered obsolete and the semantics of the `inform` PDU, initially destined for manager-manager interaction, was changed into an acknowledged notification, distribution in SNMPv2 was killed *de facto*. In SNMPv3, manager-to-agent delegation is based on the Script MIB [126], but manager-to-manager delegation still does not work because manager-manager interactions are not specified in any SNMP management architecture (v1, v2c, or v3). The few management-platform vendors that support distributed management in the IP world currently have to resort to proprietary mechanisms (e.g., HP OpenView). As a result, it is exceedingly complex to base a distributed management solution on management platforms developed by different vendors—a situation often encountered when two large enterprises merge.

To solve this problem and allow vendors to distribute management in an open and interoperable way, we decided to specify how management should be distributed in WIMA. In Chapter 3, we saw that we had two options: hierarchical and cooperative management. Which one should we choose in WIMA?

As we showed in Section 3.2.1, the topology and administration of IP networks often map directly onto the organization chart of an enterprise. The issue of distributed management arises when networks grow large, that is, when enterprises grow large. Large enterprises are generally organized in a hierarchical rather than cooperative way, so it makes sense to distribute hierarchically the management of large networks or distributed systems. Distributed hierarchical management is also easier to implement than distributed cooperative management, as we said in Chapter 4, because the technologies that support cooperative management (e.g., goal-based multi-agent systems) are still not mature enough to be used in NSM.

Therefore, in WIMA, management is distributed across a hierarchy of managers. To make the deployment of distributed management simple, we decided that manager-agent and manager-manager interactions should work similarly. As a result, everything we say about agent-manager interactions applies equally to manager-manager interactions. In particular, all the figures depicting an agent and a manager in this dissertation implicitly also depict a mid-level manager and a top-level manager. Another consequence is that in WIMA, we also have the choice between push-based and pull-based interactions to distribute management, depending on the number of times a management data is transferred from a mid-level to a top-level manager.

6.1.5 A new communication model: WIMA-CM

Our last architectural decision was to define a new communication model. This was required by the new push-based interactions between agents and managers (or between mid- and top-level managers in distributed hierarchical management) and by the dissociation between the communication and information models. Just as we had to split the organizational model into two parts, we also split the communication model into two components: one follows a push model (WIMA-CM-push) to transfer regular management data and notifications, and the other follows a pull model (WIMA-CM-pull) for *ad hoc* management data. Both of them will be described in detail in Chapter 7.

6.2 Main Design Decisions (Design Phase)

In this section, we present the main design decisions behind our proposed management architecture: WIMA.

6.2.1 Web technologies

Our first design decision was to adopt Web technologies: HTTP, HTML, XML, Java applets, Java servlets, etc. The advantages of using them in NSM are numerous. We identified eight.

First, Web technologies are ubiquitous today: they are used throughout the software industry. They are standard as long as developers follow the specifications and do not use proprietary extensions (e.g., browser-specific extensions). By going from domain-specific technologies such as SNMPv1, SNMPv2c, SNMPv3, BER, SMIv2, etc. to standard Web technologies, we make it much easier and less expensive for enterprises to find and train programmers, and we decrease the development and maintenance costs of the management application.

Second, Web technologies increase the portability of code. By coding management GUIs as Java applets instead of binaries, we dramatically reduce the development costs faced by vendors. Management GUIs no longer need to be ported to many operating systems and many management platforms offering different APIs: the same code can run anywhere, as long as proprietary Java extensions are avoided. As management GUIs are less costly for vendors, they should be less expensive for customers, too.

Third, by embedding the vendor-specific management GUIs directly in the agent, we address the issue of MIB versioning. We can have different versions of a vendor-specific MIB in the same network, and we no longer require a MIB-discovery protocol. A MIB update on the agent simply requires an update of its embedded management GUI.

Fourth, by embedding the management GUIs directly in the agent, we also bring their time-to-market down to zero. We no longer have a time lag between the availability of a new, highly performing piece of hardware and the possibility to manage it with a full-blown management GUI ported to *the* management platform and *the* operating system owned by the customer. This factor is crucial in production environments.

Fifth, by embedding the management GUIs directly in the agent, we also suppress the need for separate management platforms for start-up companies. We properly integrate the management of equipment from different vendors, which puts small and large equipment vendors in fair competition. As a result, start-ups are no longer disadvantaged by their difficult or expensive access to the APIs of the major management platforms.

Sixth, by using HTTP over TCP instead of SNMP over UDP, we simplify the management of remote subsidiaries across firewalls (we will explain this in detail in Section 7.2.2.2).

Seventh, by using HTTP instead of SNMP as a transfer protocol, we can easily compress management data, and consequently reduce the network overhead caused by management data (see Section 7.4.4).

Eighth, XML and JDBC make it easy to store management data in any third-party database (see Section 6.2.5). This frees customers from the impediment of peer-to-peer agreements between management-platform and database vendors.

6.2.2 Three-tier architecture

Our second design decision was to adopt a three-tier architecture instead of the two-tier architecture typically used by SNMP-based management platforms. The advantages of adding an application server between the client and the server are well-known and presented by many software-engineering authors [6, 78, 123, 245]. In short, they increase reusability and flexibility by allowing implementers to modify the middle tier without changing the other two. These three tiers are often given different names by different authors. Larman

[123, p. 273] calls them the presentation, application logic, and storage tiers. Fowler [78, Chapter 12] calls them the applications, domain, and database tiers. Wijegunaratne and Fernandez [245, pp. 41–78] call them the client, composite service, and data access server tiers. Ambler [6, pp. 144–148]) calls them the interface business system, business system, and business persistence system tiers, or simply client, application server, and server for short. We use Ambler's simplified terminology in our dissertation.

To adopt a three-tier architecture in WIMA, we must split the manager (that is, the monolithic, SNMP-based management platform) into three entities (see Fig. 11):

- the management station (any machine running a Web browser)
- the management server (a fixed, dedicated machine)
- the data server (a fixed, dedicated machine)

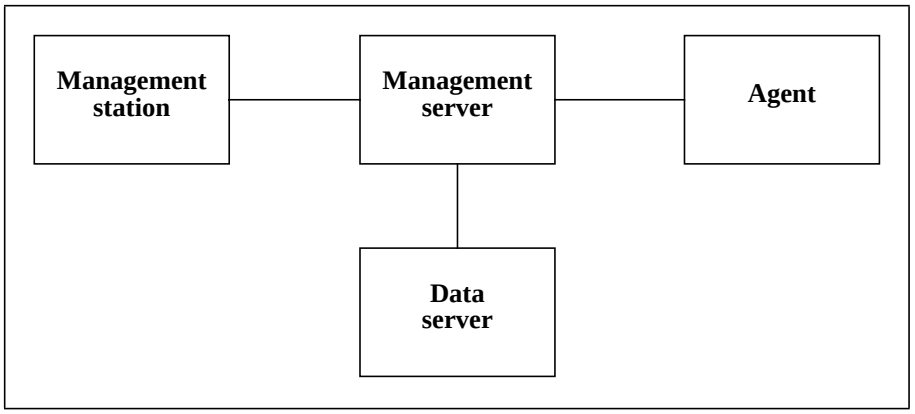

Fig. 11. Three-tier architecture

Our three-tier architecture operates with three types of interactions: between the management station and the agent, between the management and the data repository, and between the agent and the data repository.

Interactions between the management station and the agent

In interactive mode, a typical interaction goes from the management station (client), through the management server (application server), to the agent (server) and back. Such interactions typically occur in *ad hoc* management, but also happen during the subscription phase in regular management.

Interactions between the management station and the data repository

During the subscription phase, the configuration data also goes from the management station (client), through the management server (application server), to the data server (server) for persistent storage.

Interactions between the agent and the data repository

This type of interaction typically occurs during the distribution phase in regular management (data collection): data goes from the agent (client), through the management server (application server), to the data server (server). It also takes place when an agent reboots; if the agent has no local persistent storage, it can retrieve its configuration data from the data server via the management server.

We will detail all of these interactions in Section 6.3.

6.2.3 Management server: COTS components and object-oriented frameworks

Our third design decision was to improve the design of the management server by using COTS components and object-oriented frameworks instead of an opaque, monolithic, proprietary piece of code. In the 1990s, the software-engineering research community particularly studied the issue of reusability of code and design, which many researchers regarded as the main failure of object-oriented programming. They came up with a new concept to facilitate reusability: component software [222]. One outcome of this work, the component-based programming of distributed applications [107], can be very useful in NSM.

By integrating COTS components and object-oriented frameworks, we increase the competition between software vendors by making this competition more fine-grained. Today, in SNMP-based management, once you have opted for a certain management platform, you are forced to buy an event correlator, an accounting system, a billing system, or an authentication system from your management-platform vendor's catalog. You are therefore dependent on the peer-to-peer agreements signed by this vendor with third-party vendors. If you want to use a certain technology, say SecurId cards from Security Dynamics, you have to pray that your management-platform vendor has signed an agreement with Security Dynamics. Clearly, for commercial rather than technical reasons, not all technologies can be used with all management platforms; and some are available at an unreasonable price. Said otherwise, the drawback of integrated management is that customers soon become captive. Once they have invested a lot of money in a management-platform-specific solution, they can no longer drive prices down by comparing different offers from different competitors.

With a component-based management server, different parts of the management application are written by different vendors. By design, these components must be interoperable and offer open APIs to work[1]. In the component-software industry, we no longer see companies striving to keep their code opaque and proprietary to secure a niche market. COTS components and object-oriented frameworks make it possible for a customer to plug and play a new set of components to add new functionality to an existing management application, or to replace an old component by a new, smarter one. This new business model requires that existing NSM applications be restructured entirely. To follow up with the previous example, it is possible to write the security-related code in such a way that the actual security technology used by the customer is transparent to the rest of the application. Another example is the event correlator, often considered as the core of a management platform. Almost all management platforms on the market come with an event correlator. Smart event correlators are difficult to write. To make it easier for customers to work out and specify correlation rules, new languages keep appearing, either as commercial products or research prototypes. With a component-based architecture, we can plug and play a new event correlator as soon as a company releases a new smart piece of software; we no longer have to buy a brand-new management platform that necessarily comes with it.

The advantages of using component software to implement the management server are numerous. First, the customer is freed of the peer-to-peer agreements between vendors, and is no longer enchained to a single vendor by previous large investments. This drives prices down by increasing competition. It also gives customers total freedom as to what technology they can use. Second, this makes manager-manager interactions more interoperable, as third-party component vendors depend on openness for their market. As a result, distributed management solutions are easier to deploy. Third, we gain some flexibility and can change management platforms fairly easily. This is particularly useful when two companies merge and unify the supervision of their IP networks and systems.

It should be stressed that moving from current, monolithic, SNMP-based platforms to solutions based on component software is likely to significantly shake the management-platform market and cause new problems. First, new means buggy. Many years of debugging and real-life testing have gone into HP OpenView, Cabletron Spectrum, and all the major management platforms of the market. It will take years for component-based management platforms to reach the same level of reliability. Second, debugging becomes tricky. What if two components from different vendors do not interact as they are supposed to? Whose fault is it? Who should fix it? This is the usual "It's not me, it's him" situation, well-known to the administrators of

1. Note that inter-component communication is not specified by WIMA.

heterogeneous networks. To ease their task, customers can resort to software integrators, whose role is precisely to shield customers from the headaches caused by interferences between multiple vendors. But integrators reduce the cost savings of going from monolithic to component software: they have to make some profit! They also can (but do not necessarily) reduce the choice of technologies available to customers; this is a pitfall that customers should be wary about when they select a software integrator.

6.2.4 Management-data transfers across firewalls

Our fourth design decision was to take firewalls into account from the outset and to adapt our management architecture to make it firewall friendly. In WIMA, we assume that we can have one or several firewall(s) between the manager and the agent (or between the mid- and top-level managers in distributed hierarchical management). Three scenarios demonstrate why this assumption makes sense:

- An on-call administrator might want to use a PC as a management station from outside the enterprise, e.g., from home, to investigate a problem occurring in the middle of the night. To access the company's network, he/she would typically go across a firewall.
- In large, geographically dispersed enterprises, manager-manager interactions typically go across two firewalls: one at the boundary of the management domain of the top-level manager (headquarters), and another at the boundary of the management domain of each mid-level manager (subsidiaries).
- If an enterprise has a small, remote subsidiary, it can make economical sense not to buy a manager for the subsidiary, but to manage critical remote equipment across the WAN link. In this case, agent-manager interactions go across at least one firewall (the headquarters') and possibly a second (the subsidiary's).

The possible presence of one or several firewall(s) sets new constraints on our communication model. These will be detailed in Chapter 7. By taking these constraints into account in WIMA, we made it possible to use a single management architecture, whether we have a firewall or not in practice. We also addressed a shortcoming in the SNMP management architecture, which did not take firewalls into account from the outset[1].

Note that the three-tier architecture proposed in Section 6.2.2 fits nicely with firewalls, as streamlining accesses to/from agents through a single management server simplifies access control and authentication.

6.2.5 Data repository independent of the management platform

Our fifth design decision was to make the data repository independent of the management platform, thereby allowing the administrator to use the repository of his/her choice to store management data. This was one of our motivations behind the adoption of a three-tier architecture (see Section 6.2.2). To be effective, this independence must be achieved at three levels:

- the type of data repository: relational database, object-oriented database, DEN directory [214], plain-text NFS file system, etc.
- the implementation of this type, that is, the technology that underpins the data repository—e.g., an Oracle, Sybase, Ingres, or mSQL relational database
- the API used by the management application to access the data repository: XML, JDBC, Open DataBase Connectivity (ODBC), SQL strings, etc.

To achieve this loose coupling between the data repository and the management application, we assume that the data repository is logically located on a different machine called the data server (see Fig. 11). This machine may physically be different from or identical to the management server; but conceptually, we assume that the two machines are separate. Because the data repository has to be independent of the management platform, the management application can no longer embed special hooks that are specific to a given data-repository

1. To be fair with the designers of SNMP, firewalls were very rare in the late 1980s, when the SNMPv1 management architecture and communication protocol were devised.

technology (e.g., vendor-specific extensions to SQL). Instead, the management application must rely on open APIs such as JDBC, ODBC, and XML for relational databases, or XML/DEN for directories, to communicate with the data repository. By imposing the use of open APIs, we fill a void in the SNMP management architecture. The latter did not specify that proprietary APIs should be avoided to access the data repository, thereby allowing major management-platform vendors to close this market. Note that the openness of these APIs is required by WIMA, but the APIs themselves are orthogonal to our management architecture.

The independence of the management application *vis-à-vis* the data repository is very consistent with our proposed use of component software and object-oriented frameworks (see Section 6.2.3). Some components give the management application a high-level and technology-independent API to communicate with the data repository. Other components (typically, an object-oriented framework) are specific to a type of data repository. Other components might even be specific to the data repository (e.g., if we want to use the SQL extensions that are implemented by most vendors, but in different ways). Other components are specific to the API used by the management application to access the data repository.

Making the data repository independent of the management platform frees administrators from the shackles of peer-to-peer agreements between database and management-platform vendors, a problem in SNMP-based management that we described in Section 2.4.3.1. The customer can now choose any type of data repository, any implementation of this type, and any API. Another advantage is that the management application no longer has to be ported to Oracle, Sybase, Ingres, mSQL, etc. It can use a generic API such as string-based SQL for all RDBMSs, ODBC for RDBMSs running on Windows platforms, XML for all types of data repositories and all operating systems, etc. This decreases the cost of the management platform for vendors, and consequently for customers (especially with the increased competition caused by the use of component software).

6.2.6 Bulk transfers of regular management data

Our sixth design decision was to facilitate bulk transfers of regular management data, based on the assumption that by going from mere network management to full-blown integrated management, we will face a steady increase in the amount of management data to move about (see Section 2.5). In order to decrease network latency, network overhead, and CPU overhead at the manager and the agent, we want to reduce the number of messages that are exchanged between the manager and the agent in both directions [201]. We also want to prevent the manager from having to guess the size of the agent's response message when it puts together a request message (see Section 2.4.1.2 and Section 2.4.1.3). In WIMA, we meet these goals by allowing for an infinitely large answer from the agent. The engineering details of our solution will be presented in Chapter 7.

6.2.7 Dealing with legacy systems

Our seventh design decision was to take into account legacy systems in WIMA, instead of assuming that we can start all over again with a clean slate (an error initially made by Microsoft *et al.* when they issued their first WBEM proposal in 1996 [26], and later corrected by the DMTF). In Chapter 7, we prove that we can easily replace the SNMP communication protocol with HTTP. In Chapter 6, we show that we can replace the SNMP management architecture with WIMA. But to be realistic, we must be able to work with the numerous SNMP MIBs that are deployed worldwide, especially in network equipment. We must also assume that some agents do not have an embedded HTTP server, and that other agents have an HTTP server but do not have the software components that we will later propose to include in agents. Legacy systems will be dealt with in Section 6.3.6 and Section 6.4.4, when we describe migration paths from SNMP- to WIMA-based management.

6.2.8 Richer semantics

Our eighth design decision was to facilitate the use of richer semantics for management data. This will be presented in Chapter 7, when we detail the communication model of WIMA, and in Chapter 8, when we describe the advantages of using XML in integrated management. In short, the main implications are twofold.

First, because of the *my-middleware-is-better-than-yours* syndrome, we saw that we cannot base manager-agent interactions on a distributed object-oriented middleware. But this does not prevent us from using objects at all. In WIMA, we simply confine the use of Java, C++, or other objects at the edges—especially in the managers, but possibly also in the agents. This design decision proved to be useful when we developed our research prototype JAMAP (see Chapter 9). Second, with XML, we can define elaborate composite data structures that are richer, more expressive, and in the end more useful than SNMP varbind lists; we can also define many protocol operations and not only the few protocol operations defined in SNMP.

6.2.9 Easy to deploy

Last but not least, our ninth design decision was to make it easy to deploy WIMA-compliant agents and servers. The deployment simplicity was one of the keys to the success experienced by SNMPv1. With SNMP, we have learned that integrating and dealing with deployment concerns is of paramount importance to gain acceptance from the industry. WIMA is not simply a good-looking proposal on paper, it is meant to be used in real-life IP networks and systems.

This translates into many details. In particular, we integrate in this dissertation many deployment constraints that are due to the way the SNMP market historically developed, and not simply to SNMP technical constraints. We also propose a migration path to deal with legacy systems, as mentioned in Section 6.2.7. And very importantly, the amount of extra code and processing that we require the agents and managers to store and perform is reasonable. Last, we use the same communication and organizational models for agent-manager and manager-manager interactions (see Section 6.1.4.4).

Now that we have described our main architectural and design decisions, let us study in more detail the models that underlie regular management, *ad hoc* management, and notification delivery in WIMA.

6.3 Regular Management and Notification Delivery: The Push Model

The push model was recently put in the spot light by the large success encountered by push systems on the Web [96], e.g., Pointcast, BackWeb, or Tibco's TIB software suite[1]. Even though this model is well-known in software engineering, it has always been confined to notification delivery in SNMP-based management. To the best of our knowledge, no SNMP-based management platform uses it for monitoring or data collection today. Yet, as we explained in Section 6.1.4, its very design makes it better suited to regular management than the pull model that underpins SNMP-based management.

In WIMA, the push model underlies both regular management (that is, monitoring and data collection) and notification delivery. Although we commonly say that the push model is based on publish-subscribe, it actually involves four phases. Supposing that the agent supports SNMP MIBs (but the rationale remains the same with other information models), these phases are:

- *Publication phase*: Each agent maintains a list of Web pages that advertise the MIBs that it supports and the notifications that it may send to a manager.
- *Discovery phase*. The administrator finds out the URLs of these Web pages. Some of them are well known, others are vendor specific.
- *Subscription phase*: Agent by agent, the administrator (the person) subscribes the manager (the program) to different MIB variables and notifications via subscription applets. The push frequency is specified for each MIB variable.
- *Distribution phase*: At each push cycle, the push scheduler of each agent triggers the transfer of MIB data from the agent to the manager.

1. Note that despite their name, many push systems found today on the Web still follow the pull model at the implementation level. Tibco's TIB Rendez-Vous is one of the few exceptions.

In this section, we present the details of our push model: WIMA-push. We first study successively the four phases of the push model, then discuss some issues related to the distribution of management, and finally present a migration path from SNMP- to WIMA-based management.

6.3.1 Publication phase

In the first phase, each agent (or mid-level manager in distributed management) publishes:

- What information model(s) it supports (e.g., SNMP or CIM).
- For each supported information model, what virtual management-data repositories it supports. In the case of SNMP, these are the generic MIBs (e.g., MIB-II, ATM MIB, or RMON MIB) and the vendor-specific MIBs. In the case of CIM, these are the core, common, and extension schemata.
- What notifications it can send to the manager (e.g., “interface down”).

To publish this data, the agent embeds several management applets, typically stored in EPROM. These applets are available at URLs that remain constant over time for a given agent.

6.3.2 Discovery phase

In the next phase, the administrator discovers the following information for each agent:

- The information model(s) supported by the agent: SNMP, CIM, etc.
- The virtual management-data repositories supported by the agent: SNMP MIBs, CIM schemata, etc.
- The URLs of the agent’s subscription applets.

The simple way to discover all of this information is to have all equipment vendors follow the same URL naming scheme for all their embedded HTML pages and Java applets. Experience has shown that this is unrealistic, because vendors have a natural tendency to do things their own way. A similar problem shows up in a different but related area: vendor-specific MIBs. A comparison between the proprietary MIBs of the main equipment vendors highlights a remarkable heterogeneity: they all have different structures and follow different naming schemes. In other words, the discovery phase cannot be fully automated.

Conversely, if we have no automation whatsoever—that is, if the URLs where the agent publishes its subscription applets are totally random and unpredictable—, we have to manually configure the manager, agent by agent. In practice, this means that an operator must type in a multitude of URLs for all the subscription applets of all the agents... Clearly, this is not an option either because it does not scale and is error prone. In short, the discovery phase must be partially automated.

In this section, we propose a scheme whereby some URLs are well-known, because they follow a convention adopted by all vendors, and other URLs are discovered via interactive browsing. Browsing requires a starting point, which we call the agent’s management home page. Let us describe how the administrator accesses it.

6.3.2.1 A well-known URL for the agent’s management home page

In order for the administrator to access the agent’s management home page, the manager must know what URL to download. We have five options:

- The manager is manually configured by an operator, for all the agents in its management domain.
- There is an automated way for the agent to tell the manager what the URL of its management home page is.
- There is an automated way for the manager to retrieve this URL from the agent.
- There is an automated way for the manager to retrieve this URL from an external data repository.
- The manager can build this URL automatically because it follows a convention adopted by all vendors.

The first option is tedious and does not scale. We rule it out.

The second option is based on notifications. Whenever the URL of its management home page is changed, the agent sends the new URL to the manager via a notification. This scheme is not robust if implemented with SNMP notifications, because they are not acknowledged (SNMPv3 `inform`'s are not used in practice). It does not work with CIM events either, because they are not yet standardized. If we decide to use SNMPv3 `inform`'s, which are acknowledged, we will still face a major problem: deployment. This new notification must be defined in a MIB and supported by deployed agents, which requires extending an existing MIB (notifications are usually defined in vendor-specific MIBs) or defining a new MIB. But deploying a new MIB takes a lot of time; so does the upgrade of an already deployed MIB. And even worse, new MIBs are not necessarily adopted by the market, sometimes for nontechnical reasons. This solution is too risky: we have to exclude it.

The third option cannot rely on Web technologies[1]. It requires that the URL of the agent's management home page be stored in a virtual management-data repository on the agent. This repository can be an SNMP MIB, a CIM schema, etc. In the case of SNMP, the URL of the agent's management home page can be encoded in a string and stored in a MIB. This solution requires either extending an existing SNMP MIB supported by most existing agents (e.g., MIB-II), or defining a new MIB and convincing all vendors to support it (e.g., see HP's approach with the HTTP Manageable MIB [94]). This poses three problems. First, and justifiably so, the IETF has always been reluctant to change existing and widely deployed MIBs for obvious compatibility issues. Second, deploying or upgrading a MIB is time consuming and hazardous (see previous option). Third and worst, this approach would tightly tie Web-based management to a given information model (SNMP in our example, but it could be CIM as well). We do not want this, as one of the requirements for WIMA is that it should not rely on a specific information model (see Section 6.1.3). We therefore eliminate this option.

The fourth option combines the previous two. It requires that the URL of the agent's management home page be stored in an external data repository, that is, a machine that is neither the manager nor the agent. The agent stores (and possibly updates) this URL in the repository, and the manager accesses this information in read-only mode. This is the approach adopted by Sun Microsystems with its Java-based "universal" plug-and-play technology: Jini [216]. This data repository can be of different types (see Section 6.2.5); presently, the typical way of storing such configuration data is to use a DEN directory. Although this solution works, we did not retain it for WIMA because there is no consensus as yet as to how DEN directories should be designed, structured, etc. Different vendors support different schemes, and it is unclear how the situation will evolve. Another, serious problem with this approach is that it poses security problems. We need some external, untrusted agents to access and store data into an internal, trusted data repository without using strong security (see Section 2.5, "Security and firewalls")... A security officer's nightmare! We will address security issues in more detail in Chapter 7.

In WIMA, we selected the fifth option[2]. The agent's management home page must be published at a well-known URL. Let us describe our URL naming scheme.

6.3.2.2 URL naming scheme

The agent's management home page is not the only URL that must be well known. In Section 6.3.3 and Chapter 7, we will see that several other URLs must be well known. In this section, we define a general URL naming scheme that specifies (i) a number of well-known URLs supported by all vendors, and (ii) a convention for finding out proprietary URLs via interactive browsing.

To define a well-known URL, we can impose a certain pathname, a certain port number, or both. We need both. If we simply impose a pathname, we can for instance reserve the keyword `mgmt` for "management". The well-known URL for the agent's management home page is then:

```
<http://agent.domain/mgmt/>
```

1. Otherwise, it would lead to an infinite loop. To publish the URL of its management home page, the agent would have to publish where it publishes it. By recurrence, it would have to publish where it publishes where it publishes it, etc.
2. Note that once DEN has made progress at the DMTF and the way to store an agent's management home page has been defined, it will be very easy to change slightly WIMA to support the new standard DEN directory.

where `agent.domain` is either the fully qualified domain name or the IP address of the agent. The problem here is that there are already many Web servers running all over the planet, some in embedded equipment (hence difficult to upgrade), and the odds are that some of them already use the reserved pathname `mgmt` for other purposes. In fact, it is impossible to find a pathname that we know for sure is not used yet. In consequence, imposing a certain pathname is not sufficient.

Alternatively, we can impose a port number. This approach was proposed by Harrison *et al.* [94] in 1996. They recommend the use of the reserved TCP port 280 (`http-mgmt`) for Web-based management:

```
<http://agent.domain:280/>
```

By not using the default port 80, and by using instead a port that is dedicated to Web-based management, we significantly reduce the risks of URL clashes. Even though port 280 is seldom used today, it is feasible to make all vendors agree to use it. However, this is not enough. Selecting a well-known port is sufficient for retrieving a single URL, e.g. the agent's management home page. But what about the other well-known URLs?

We really need to impose both the port number and the pathname. By doing so, we get the best of both worlds: we avoid URL clashes by not using the default port 80, and we keep the flexibility offered by fixed pathnames to automate management. As port 280 is dedicated to Web-based management, it is now possible to make all vendors agree on a few basic path-naming conventions.

Note that in practice, an agent may run its embedded HTTP server on any port, as long as the configuration file of its HTTP server is able to map all incoming requests destined for port 280 to the actual port that its embedded HTTP server is listening on (e.g., port 80). In fact, this trick allows a vendor to support the two ports (80 and 280) simultaneously, e.g. for backward compatibility.

The naming convention that we propose in WIMA for the well-known URLs is the following. The management home page of an agent is served by its embedded HTTP server at the following URL:

```
<http://agent.domain:280/mgmt/>
```

where `agent.domain` is the fully qualified domain name or the IP address of the agent. We keep the pathname prefix `/mgmt/` (although it is redundant, since port 280 is dedicated to management) so as to avoid clashes with the few existing uses of port 280.

On its management home page, the agent publishes the URLs of different pages that describe the different management architectures and information models that it supports. The format of these pages is free. Examples of pages for multiple management architectures include:

```
<http://agent.domain:280/mgmt/arch/wima/>
<http://agent.domain:280/mgmt/arch/wbem/>
<http://agent.domain:280/mgmt/arch/jmx/>
<http://agent.domain:280/mgmt/arch/fma/>
```

Examples of URLs for multiple information models include:

```
<http://agent.domain:280/mgmt/infomodel/snmpv1/>
<http://agent.domain:280/mgmt/infomodel/snmpv2c/>
<http://agent.domain:280/mgmt/infomodel/snmpv3/>
<http://agent.domain:280/mgmt/infomodel/cim-spec-2.0/>
<http://agent.domain:280/mgmt/infomodel/cim-spec-2.2/>
<http://agent.domain:280/mgmt/infomodel/osi/>
```

where `cim-spec` refers to the DMTF expression *CIM Specification*. In general, we are not very interested in general information about the information models supported by an agent, but rather in getting a list of all the

virtual management-data repositories (SNMP MIBs, CIM schemata, etc.) supported by this agent. These are published at the following URLs:

```
<http://agent.domain:280/mgmt/infomodel/smiv1/>
<http://agent.domain:280/mgmt/infomodel/smiv2/>
<http://agent.domain:280/mgmt/infomodel/cim-schema-2.0/>
<http://agent.domain:280/mgmt/infomodel/cim-schema-2.2/>
<http://agent.domain:280/mgmt/infomodel/cim-schema-2.3/>
<http://agent.domain:280/mgmt/infomodel/cim-schema-2.4/>
```

where `cim-schema` refers to the DMTF expression *CIM Schema*, and versions 2.0 through to 2.4 are the four versions currently supported by the DMTF (previous versions have been rendered obsolete). Note that in the case of SNMP, the semantics of the MIBs depends on the metamodel (SMIv1 or SMIv2) rather than the SNMP management architecture (SNMPv1, SNMPv2c, or SNMPv3) for historical reasons. So, for instance, the following URL lists all the SNMP MIBs expressed in SMIv2 that are supported by the agent (or mid-level manager):

```
<http://agent.domain:280/mgmt/infomodel/smiv2/>
```

When a manager wants to discover the preferred version of SNMP or CIM supported by an agent, it requests a generic URL such as:

```
<http://agent.domain:280/mgmt/infomodel/snmp/default.html>
<http://agent.domain:280/mgmt/infomodel/smi/default.html>
<http://agent.domain:280/mgmt/infomodel/cim-spec/default.html>
<http://agent.domain:280/mgmt/infomodel/cim-schema/default.html>
```

Whenever the agent receives a request for a generic URL, its HTTP server replies with a `Redirection` status code of 301 (Moved Permanently) and a `Location` field indicating its preference, e.g.:

```
<http://agent.domain:280/mgmt/infomodel/snmpv3/>
<http://agent.domain:280/mgmt/infomodel/smiv2/>
<http://agent.domain:280/mgmt/infomodel/cim-spec-2.2/>
<http://agent.domain:280/mgmt/infomodel/cim-schema-2.4/>
```

Similarly, a manager can discover the preferred information model of an agent by requesting the following URL:

```
<http://agent.domain:280/mgmt/infomodel/default.html>
```

This is useful in transition phases, for instance in case CIM gradually takes over SNMP in the years to come. The agent then redirects the manager (with an HTTP status code of 301) to the home page of its preferred information model. If it has no preference, it returns the home page of any supported information model.

The same rationale applies to the management architecture. A manager can discover the preferred management architecture of an agent by accessing the following URL:

```
<http://agent.domain:280/mgmt/arch/default.html>
```

6.3.2.3 From the network map to the agent's management home page

Now that the manager knows the URLs of the management home pages of all the agents in its management domain, it still has to present this information to the administrator for the subscription phase to take place. We propose that direct access to the agent's management home page be available directly from the network map. This can be achieved in different ways, depending on whether the network map is implemented in the form of a plain HTML page, a sensitive map, or a Java applet.

With a plain HTML page, we simply display a textual list of agents in the management domain. Each line corresponds to an agent and contains a hyperlink to this agent's management home page. This approach is minimalist and not user friendly, but it works and can be sufficient to manage small networks.

When the network map is a sensitive map, e.g. a Graphics Interchange Format (GIF) image, the operator clicks on the icon representing the agent. The (x,y) coordinates are mapped to the corresponding agent by a CGI script running on the management server. The output of this CGI script is an HTTP response message with a `Redirection` status code of 301 (Moved Permanently) and a `Location` field containing the URL of the corresponding agent's management home page. Upon receipt of this redirection, the browser automatically contacts the agent and retrieves its management home page. The mapping table used by the CGI script can be generated automatically: from the agent's IP address, it is possible to generate the URL of its management home page.

When the network map is coded as a Java applet, we can have an even more user-friendly GUI. For instance, by clicking on the agent on the network map, an operator can pop up a menu and select the entry called "home page" to retrieve the agent's management home page. One problem though is the applet security model, which by default forbids an applet loaded from the management server to open a connection to another machine, in our case the agent. To work around this limitation, we can use signed applets. Although this solution is simple and neat, it is not always applicable because it requires the Web browser to access a configuration file specifying its access-control policies, and this clashes with our decision to enable the administrator to use a Web browser on *any* machine. There are means to share a configuration file across many machines, e.g. NFS in the Unix world; but there are cases when we need to duplicate the configuration file (e.g., when an on-call administrator troubleshoots an urgent problem from outside the enterprise). In short, a network-map GUI applet is very user-friendly, but there are cases when we need to revert to sensitive maps.

6.3.3 Subscription phase

In the third phase, the administrator downloads the agent's subscription applets into the management station (see Fig. 12) and explicitly subscribes the manager to the regular management data (e.g., SNMP MIB variables or CIM objects) and notifications (e.g., `SNMPv2-trap`'s or CIM events) that he/she is interested in. For the sake of readability, let us assume that the agent supports an SNMP information model (v1, v2, or v3), although everything we say here for SNMP applies equally well to other information models. We distinguish two cases: the interactive mode, where the administrator interacts via a Web browser, and the automated mode, where everything is done automatically by programs without human intervention.

6.3.3.1 Interactive, human-oriented mode

In interactive mode, the administrator selects the management applet he/she is interested in from the agent's management home page. For the subscription phase, two entries are of interest to us: the data subscription applet and the notification subscription applet.

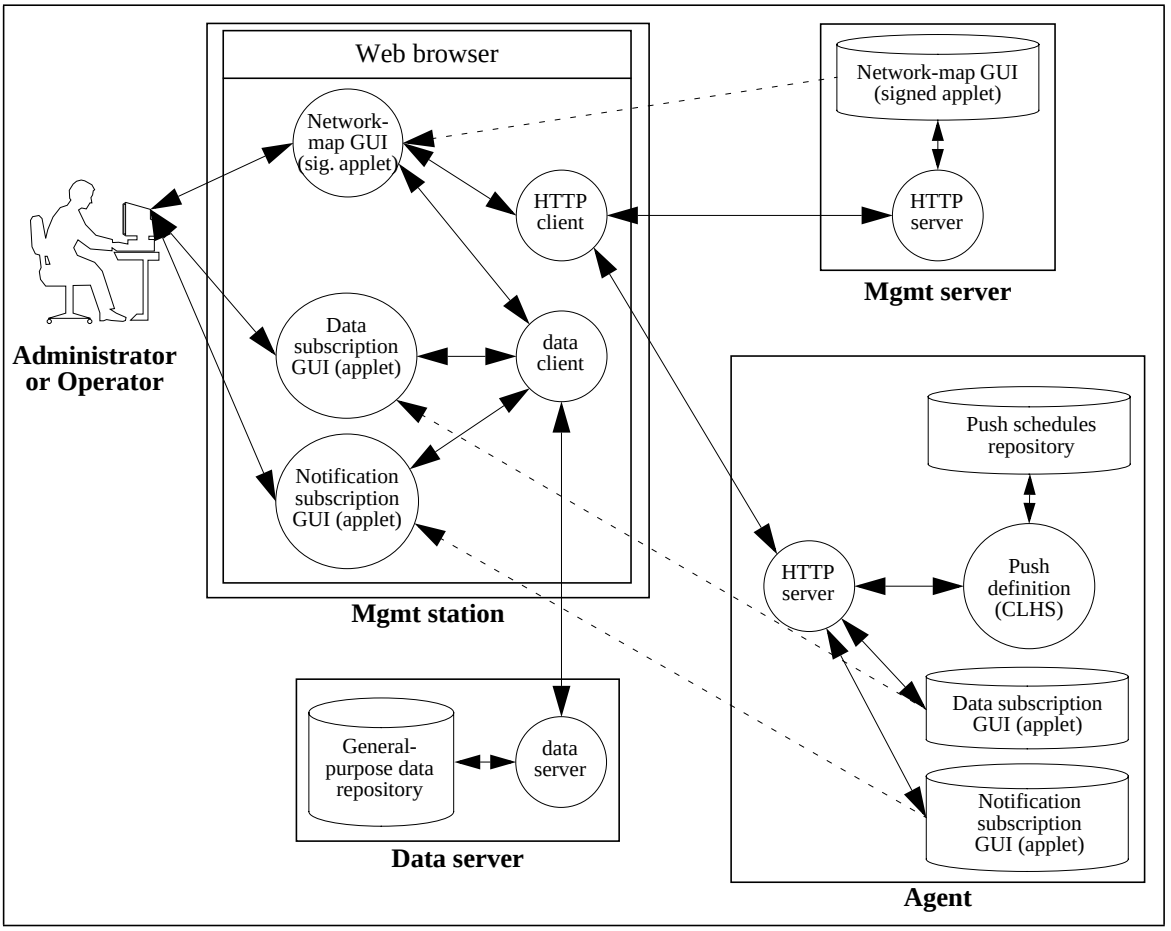

Fig. 12. Push model: discovery and subscription phases (without firewall)

The *data subscription applet* allows the administrator to interactively select MIB variables and push frequencies via a user-friendly GUI. This is commonly achieved through some kind of MIB browser. The push frequency can be different for each individual MIB variable, or it can be the same for an entire MIB or even all the MIBs supported by a device. In the case of SNMP, this applet is called the *MIB data subscription applet*. We will come back to this applet in more detail in Chapter 9, when we describe our prototype.

The *notification subscription applet* allows the administrator to interactively select the notifications that he/she wants the manager to receive. In essence, it defines an event filter on the agent. If we have just a few notifications to filter, the full-blown applet can be replaced with a simple HTML form. Notifications for which the administrator showed no interest are discarded by the agent. Unlike its counterpart, the notification subscription applet does not prompt for a push frequency because notifications are inherently asynchronous.

The retrieval of these two subscription applets is illustrated by Fig. 12. The dotted arrows are visual aids that show that the corresponding applet is transferred from one machine to another. The plain arrows indicate that two components communicate directly with each other.

We can have one subscription applet for all information models (but this might not be practical), one per information model (e.g., SNMP or CIM), or one per virtual management-data repository (e.g., one per SNMP MIB or CIM schema). The same rationale applies to notifications. The granularity of the two subscription applets is vendor specific and not mandated by WIMA. Regardless of the granularity, the MIB variables subscribed to through the same GUI and their associated frequencies form what we call a *push schedule*. Push schedules are created by the *push definition* component. This component can be implemented as a Java servlet, a CGI binary, a CGI script, etc. To remain generic, we will call it a *Component Launched by an HTTP Server* (CLHS). This naming convention will be used throughout this dissertation for similar components. If possible,

the push schedules are stored persistently by the agent (e.g., in EPROM). Otherwise, they are kept in volatile memory and must be retrieved from the data server (via the management server) upon reboot; we will see how in Section 6.3.3.3.

For the SNMPv1, v2c, and v3 information models, the URL of the data subscription applet is one of the following:

```
<http://agent.domain:280/mgmt/subscribe/smiv1/mibs.html>
<http://agent.domain:280/mgmt/subscribe/smiv2/mibs.html>
```

In the case of CIM, the URL of the data subscription applet simply depends on the version of the CIM schemata supported by the agent. Examples of valid URLs include:

```
<http://agent.domain:280/mgmt/subscribe/cim-schema-2.2/schemata.html>
<http://agent.domain:280/mgmt/subscribe/cim-schema-2.3/schemata.html>
<http://agent.domain:280/mgmt/subscribe/cim-schema-2.4/schemata.html>
```

For each version of the CIM schema, it is possible to distinguish between the three sets of schemata defined by the DMTF: core, common and extension. In the case of CIM Schema 2.3, this yields:

```
<http://agent.domain:280/mgmt/subscribe/cim-schema-2.3/core.html>
<http://agent.domain:280/mgmt/subscribe/cim-schema-2.3/common.html>
<http://agent.domain:280/mgmt/subscribe/cim-schema-2.3/extension.html>
```

The URLs of the notification subscription applets look like this:

```
<http://agent.domain:280/mgmt/subscribe/snmpv1/traps.html>
<http://agent.domain:280/mgmt/subscribe/snmpv2/notifications.html>
<http://agent.domain:280/mgmt/subscribe/cim-schema-2.4/events.html>
```

Note that this time, SNMP traps and notifications depend on the version of SNMP rather than SMI, because SNMPv2 rendered SNMPv1 traps (`trap` PDUs) obsolete and replaced them with SNMPv2 notifications (`SNMPv2-trap` PDUs), which remained unchanged in SNMPv3. This inconsistency in our URL naming conventions is unfortunate, but alas reflects the history of SNMP.

In interactive mode, we only ask that an agent support a well-known URL for its management home page. From there, the administrator can navigate to find the SNMP MIB or the notifications he/she is interested in, select some MIB variables or notifications, and submit the subscription to the agent. But in real networks, the number of devices and systems to manage can be large. In this case, manual configuration is not only tedious and error prone, but also totally unrealistic. As a consequence, our management architecture must not necessarily rely on interactive configuration. We want to have the option to automate this subscription phase, by cloning the push schedules of similar agents and sending them automatically from the management server to the agents.

6.3.3.2 Automated, machine-oriented mode

In automated mode, the situation is not as simple because a manager is not as smart as an administrator. Its semantic analyzer is very weak compared to the administrator's brain, so we cannot expect it to navigate from the agent's management home page to find the pages to subscribe to MIB variables or notifications. If we want to automate this process, the two interactive subscription applets must have two machine-oriented counterparts (the *subscription* CLHS) at well-known URLs. The manager can then use these subscription CLHSs to send a predefined set of push schedules and notification filters to each agent.

The URLs of the data subscription CLHS(s) supported by the agent look like this:

```
<http://agent.domain:280/mgmt/subscribe/auto/smiv2/mibs>
<http://agent.domain:280/mgmt/subscribe/auto/cim-schema-2.4/schemata>
```

Note the term `auto` in the pathname. Likewise, examples of URLs for the notification subscription CLHS include:

```
<http://agent.domain:280/mgmt/subscribe/auto/snmpv1/traps>
<http://agent.domain:280/mgmt/subscribe/auto/snmpv2/notifications>
<http://agent.domain:280/mgmt/subscribe/auto/cim-schema-2.3/events>
```

For automated subscriptions, the absence of an extension at the end of the pathname allows for any type of subscription CLHS: CGI script, CGI binary, Java servlet, etc. This gives us a lot of flexibility.

6.3.3.3 Data repository

If an agent is not able to store its subscription data (push schedules and notification filters) in EPROM, it loses its configuration upon reboot. In interactive mode, it would be very tedious for the administrator to re-enter all the subscription data of an agent whenever this agent reboots. Therefore, it is important to store this data in persistent storage. Similarly, in automated mode, we need a persistent data repository to retrieve the push schedules and notification filters, clone them from a similar device, and send them to the agent (for the first time or if it has lost its subscription data).

As shown by Fig. 12 and Fig. 13, we use the data repository for storing subscription data. For the sake of readability, the details of the different repositories are not shown on this figure. All repositories are virtually merged into a single *general-purpose data repository* that includes:

- the push schedules for the management data subscribed to by the manager
- the notifications subscribed to by the manager
- the network topology used by the network-map GUI applet to construct its GUI

In practice, these three logical data repositories may be physically stored into one or several relational databases (RDBMSs), object-oriented databases (OODBMSs), NFS servers (flat files), DEN directories, etc.

If an agent loses its subscription data, the restoration procedure that intuitively comes to mind is the following:

- the agent asks the manager for its subscription data
- the manager retrieves the agent’s push schedules and notification filters from the data server
- the manager sends the push schedules and notification filters to the agent

We will see in Chapter 7 that, for security reasons, the agent cannot inform the manager that it has just rebooted: the manager must detect it. The procedure to follow must therefore be slightly changed:

- the manager detects that the agent has just rebooted
- the manager retrieves the agent’s push schedules and notification filters from the data server
- the manager sends the push schedules and notification filters to the agent

6.3.3.4 Subscription with firewall

In Section 6.3.3.1, we considered a simplified case without firewall between the agent and the manager (see Fig. 12). The manager can then directly interact with the agent (two-tier architecture); this poses no security threat. This scenario is typically encountered in the management of small intranets, where the internal network is trusted. But in Section 6.2.4, we explained that we want to allow for the presence of a firewall between the agent and the manager. So let us now study the complete scenario with a firewall (see Fig. 13). This scenario

applies equally well to the interactive and automated modes, with one exception: in automated mode, there is no communication with the management station.

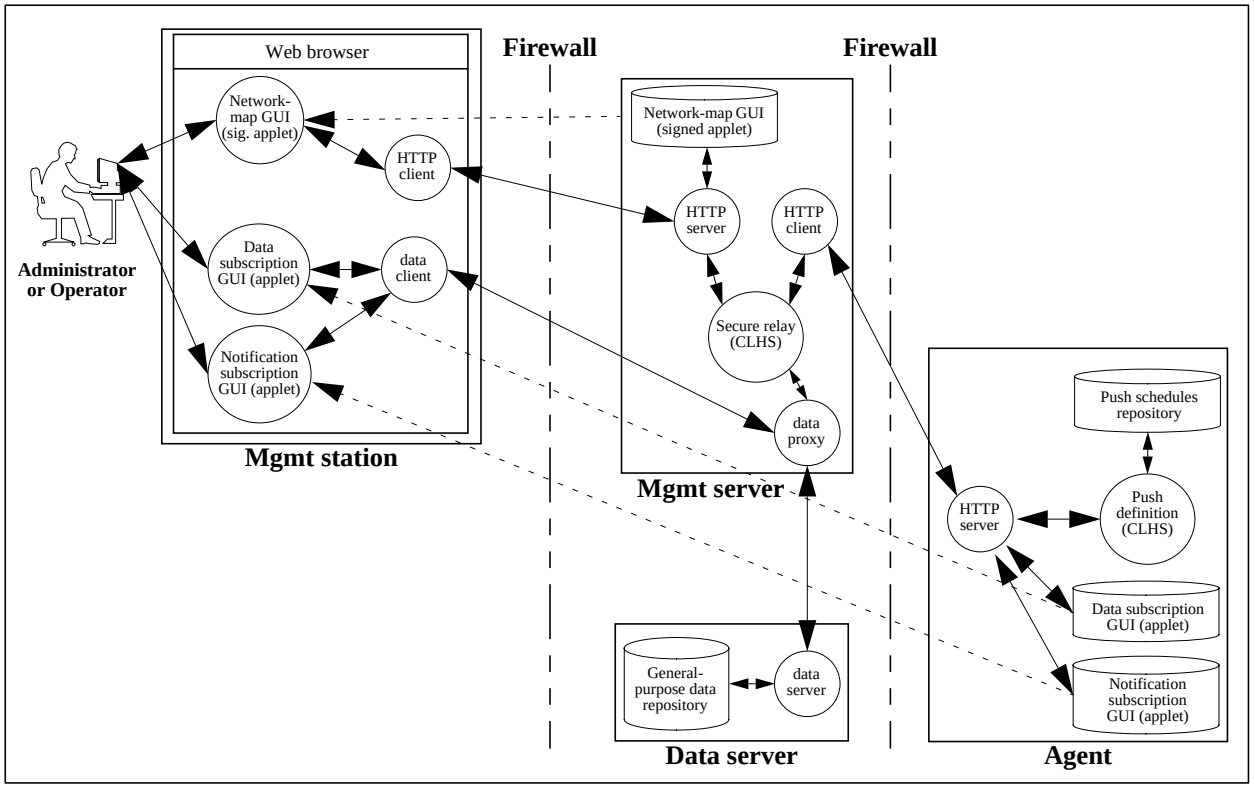

Fig. 13. Push model: discovery and subscription phases (with firewall)

The firewall system typically allows external agents to interact only with one internal machine (the management server), as opposed to many internal machines (all the potential management stations, that is, all internal PCs and workstations). So, in the firewall case, the communication between the management station and the agent must follow a three-tier architecture and be streamlined via the management server. This is achieved by the *secure relay* CLHS, which forwards unchanged all the HTTP traffic from the internal management station(s) to the external agent(s), and *vice versa*. The role of the management server is then fairly similar to that of an HTTP proxy in the World-Wide Web. The internals of the secure relay are not specified by our management architecture. This CLHS uses access-control lists stored in a data repository that can be physically separate from or integrated with the management server.

One advantage of this setting is that it allows the administrator to monitor the network or take corrective actions from outside the enterprise (see Section 6.2.4). In this case, the communication goes across a firewall, which can be similar to or different from the firewall located between the manager and the agent. The identification and authentication (not represented here) are performed similarly by the secure relay, whether the administrator works from an internal or an external machine. Obviously, direct access to the agents from external management stations must be forbidden for security reasons.

6.3.4 Distribution phase

In the last phase, the case of data collection and monitoring is only marginally different from that of notification delivery in WIMA. The organization and communication models are the same. Only the CLHSs and object-oriented components running on the agent and management server (or on the mid-level and top-level managers) are different. Let us begin with notification delivery, which is slightly simpler.

6.3.4.1 Notification delivery

Notification delivery from the agent to the management server is depicted in Fig. 14, together with event handling within the management server. Event handling will be studied in Section 6.3.4.3.

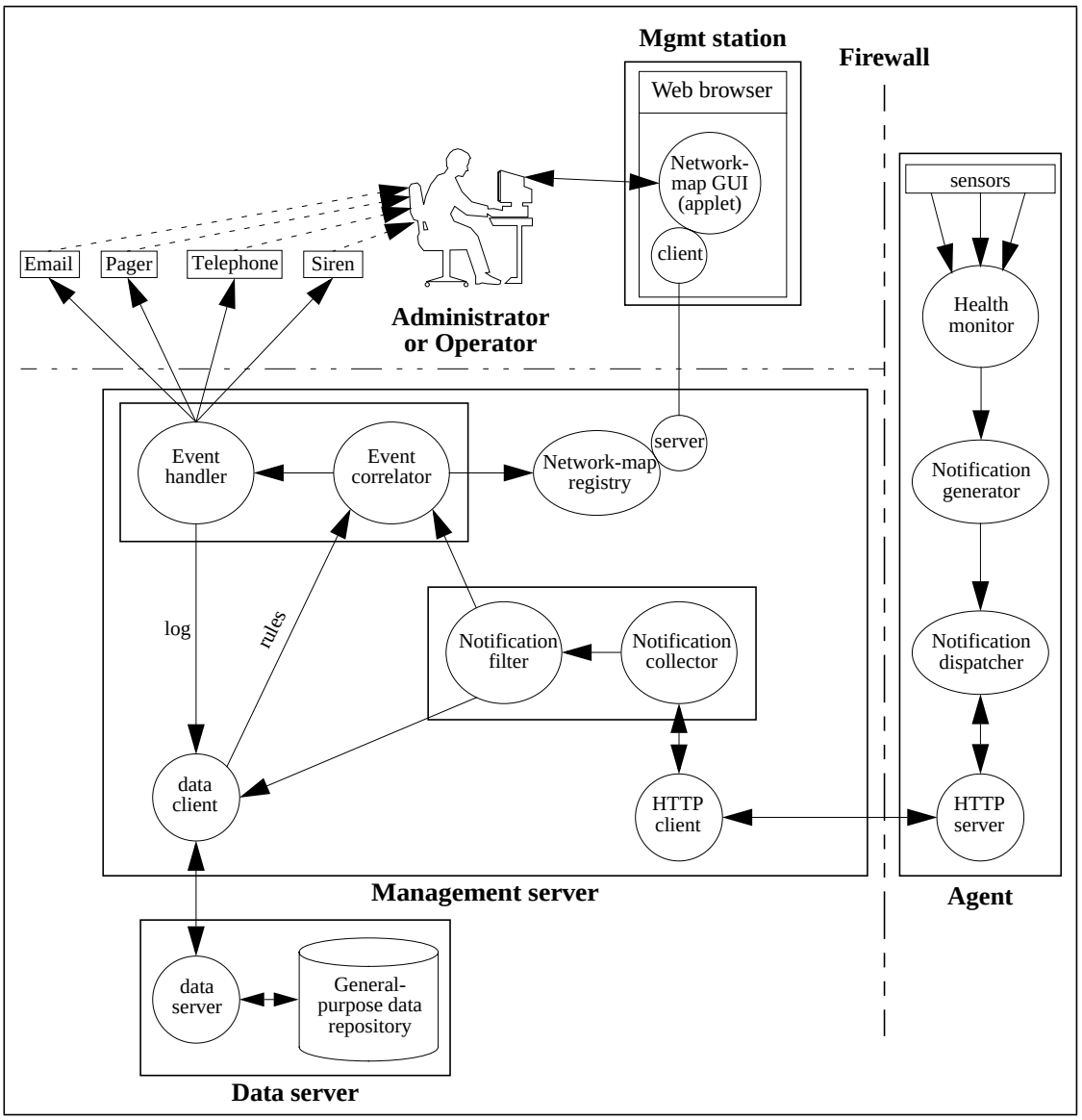

Fig. 14. Push model: distribution phase for notification delivery and event handling

The agent runs a *health monitor* that checks the agent's own health. This component receives input from a number of *sensors* that can be implemented in hardware or software. For instance, the health monitor checks whether the agent's Ethernet interfaces all sense a carrier, or whether the ventilation of the power supply is still working. (Note that in the IP world, agents are usually not able to monitor many things, unlike the agents typically found in the telecom world.) When a problem occurs and is detected, the health monitor sends an

alarm to the *notification generator*. The notification generator translates this alarm (a vendor-specific data structure) into a standard SNMP notification, CIM event, and so on, depending on the information model(s) supported by the agent and on the type of notifications subscribed to by the administrator. This notification is then marshalled and sent by the *network dispatcher* to the management server. For the time being, we skip the communication path between the agent and the management server (that is, the question mark on Fig. 14); we will describe it in Chapter 7.

When it arrives on the management server, the notification is handled by the *notification collector*, which unmarshalls it. The notification collector passes this notification on to the *notification filter*, whose role is to protect the management application from misconfigured, misbehaving, or malicious agents. If the manager is bombarded with notifications by an agent, this filter silently drops them, tells the notification collector to close the connection to the agent (this connection will be defined in Chapter 7), logs a single entry in the data server, and sends a single event to the event correlator. If the administrator previously set up an event handler for that type of event, he/she can be informed immediately that something is wrong with this agent. Once the notification filter has checked an incoming notification, it sends it to the *event correlator*. As in SNMP-based management platforms, the event correlator is the focal point of monitoring in WIMA. We will come back to it in Section 6.3.4.3, once we have described another important source of events.

6.3.4.2 Data collection and monitoring

Let us now consider data collection and monitoring. As depicted in Fig. 15, the organizational model remains the same, and the building blocks look alike. The two main differences between this case and the previous one lie in the components and CLHSs running on both sides and in what triggers a management-data transfer.

When the push scheduler determines that it is now time for the next push cycle, it contacts the *data formatter* and tells it what SNMP MIB variables, CIM objects, etc. to send to the manager. The data formatter retrieves the requested data from the virtual management-data repositories (SNMP MIBs, CIM schemata, etc.), formats these vendor-specific data structures into standard SMIv2 entities, CIM objects, etc. and sends them to the *data dispatcher*. The latter marshalls this data and sends it to the management server. Again, we postpone the description of the communication path between the agent and the manager until Chapter 7. This time, on the management server, the data is received by the *pushed-data collector* which acts just as the notification collector (we will see in Chapter 7 why we preserve two separate processing units, instead of merging them). The data is then unmarshalled and sent to the *pushed-data filter*, which increases the robustness of the management application (same function as the notification filter). If this filter is happy with the rate at which data is coming in from this agent, it passes on the data to an entity which has no equivalent in notification delivery: the *pushed-data interpreter*. At this level, the data forks off, depending on whether it is related to monitoring or data collection.

For data collection, incoming data is not analyzed directly: it is stored into the data repository so as to be analyzed offline, at a later stage. The pushed-data interpreter sends incoming data to the data repository via some kind of technology orthogonal to WIMA (e.g., JDBC, ODBC, or NFS). The client-server nature of this storage operation is simply denoted in Fig. 15 with the *data client* component on the management server and the *data server* component on the data server machine.

For monitoring, the data must be analyzed immediately, as it comes in. To achieve this, the pushed-data interpreter executes a set of *rules* not represented on Fig. 15. Together with the event correlator, the pushed-data interpreter constitutes the smartest part of the management application. In the IP world, the pushed-data interpreter makes up for the lack of self-management functionality in agents (unlike the agents typically found in the telecom world). The rules are persistently stored in the data server, and dynamically cached by the management server. They allow the pushed-data interpreter to check that everything is “normal”. When the pushed-data interpreter detects a problem, it sends an event to the event correlator—for instance, when a network device no longer sends a heartbeat. The pushed-data interpreter typically interacts with the finite state machines that represent each agent in the management application, and determines whether a

problem is transient or semi-permanent. Transient problems are usually ignored, whereas semi-permanent problems are dealt with by event handlers, operators, or both (see next section).

If a piece of data (be it related to monitoring or data collection) does not come in on schedule, the pushed-data interpreter generates an event and sends it to the event correlator. This requires the management server to run a scheduler component that is not depicted in Fig. 15.

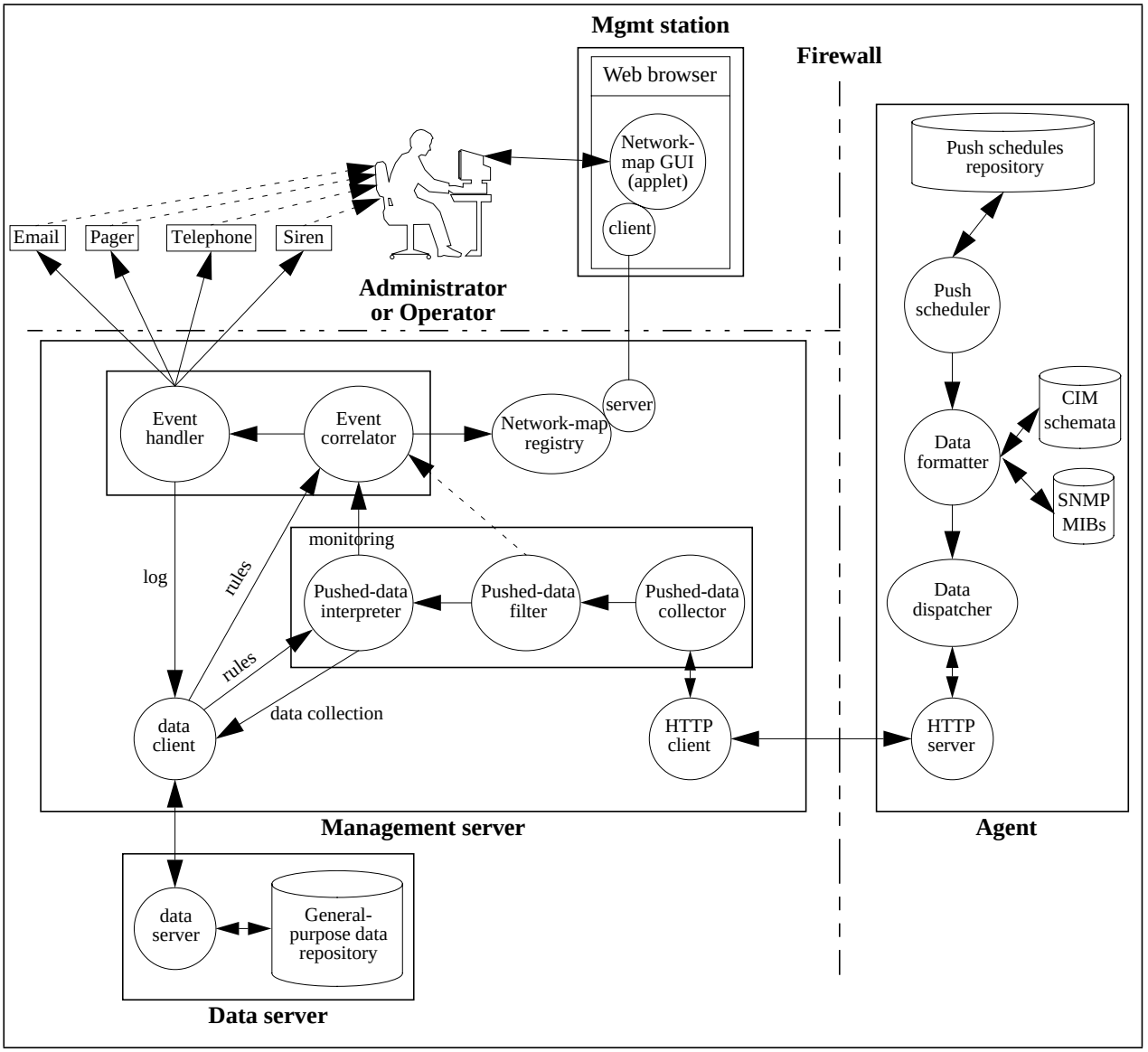

Fig. 15. Push model: distribution phase for data collection and monitoring

6.3.4.3 Event handling

In the previous two sections, we have seen two sources of events: the agents, which send notifications when they are able to detect problems by themselves, and the pushed-data interpreter, which detects problems on behalf of the agents. The event correlator is situated at the intersection of these two flows of events, that is, between regular management and notification handling. It is the “clever” part of the management application: it is able to find out the root cause when an avalanche of problems is reported for one or several machine(s). For instance, if a router is hung, the hosts behind that router will appear to be down, but no corrective action should be taken for the hosts: only the router should be rebooted. Working this out is the role of the event correlator.

Among all the events that are processed by the event correlator, some are masked (e.g., those related to the hosts in the previous example) and silently discarded; others do not require any action (e.g., informative events); yet others trigger some action (e.g., the root-cause event in the previous example). So far, event handling is similar in WIMA- and SNMP-based management platforms. The difference is the way actions are invoked. We distinguish two types of actions (informative and corrective) and three types of mode (interactive, fully automated, and semi-automated).

Informative actions simply aim at informing operators or the administrator that a problem occurred. Whether something should be done in response to this problem is left for a human to decide. Conversely, corrective actions attempt to automatically solve a problem. Typically, low-severity events always trigger informative actions (nothing to do immediately), medium-severity problems sometimes trigger corrective actions and sometimes trigger informative actions (not all problems can be solved automatically), and high-severity events always trigger informative actions (too difficult to address automatically).

In interactive mode, at least one administrator or operator must register a network-map GUI with the *network-map registry* to be told about informative actions. In this case, the event correlator forces an update of all the registered GUIs when the finite state machine of an agent changes its state. Typically, the icon depicting that agent turns red, green, or yellow to reflect the new state of that agent (respectively `problem`, `OK`, or `unknown`). If the network-map GUIs display the network topology in a hierarchical manner, the icons representing the topological hierarchy of this agent will also change their color accordingly.

In fully automated mode, no network map is registered with the network-map registry. NSM is then entirely dependent on *event handlers* defined by the administrator, typically components in WIMA. The severity level of an event determines the event handler that is invoked by the event correlator. For informative actions, it also determines the emergency mode used to inform the administrator: an event handler might simply log the problem in the data repository, or send an email to the administrator, or take more drastic actions such as paging the administrator or starting off a siren. The way event handlers are configured by the administrator is intrinsically site specific and totally independent of WIMA.

In real life, management platforms are often configured to operate in semi-automated mode. This means that some problems are solved automatically by the management application (corrective actions) while others are not (the operators monitoring the network are simply notified about informative actions). For instance, if a central IP router crashes in an intranet, an icon will turn red on the network-map GUI to prompt the operators to take some action (e.g., reboot it, or replace a faulty board). But as other routers take over the failed one, all hosts with preexisting connections are temporarily hung because they have obsolete entries in their Address Resolution Protocol (ARP) cache. One way to alleviate this problem is to automatically clear the ARP cache of all the IP routers of the intranet, as soon as an IP router is deemed to have crashed. In this scenario, the management application clearly takes both informative and corrective actions at the same time.

When an event handler is invoked, it makes sense to log an entry in the data repository. But we must be careful not to run into scalability problems. For instance, when the IP networks and systems being managed are not very stable, the event correlator might have to process a continuous flow of incoming events, and might as a result continuously invoke event handlers. Administrators should therefore be careful not to flood the data repository with too much information. Most of the time, only *some* events should be logged in the data repository. It is often more useful to log event statistics rather than the actual events. The computation of event statistics may be achieved by components running on the management server, and invoked by the event handlers (that then no longer invoke directly the *data client* component). Similarly, icons should not turn too often red or green on the network-map GUI, otherwise operators would not know what to do. Such oscillations can easily be reduced by adding some dampening in the finite state machines (e.g., it might take at least a minute to go from green to yellow, and another minute to go from yellow to red).

6.3.4.4 Data repository

Compared with Section 6.3.3.3, we have added a number of repositories for the distribution phase. The general-purpose data repository now includes nine repositories:

- the push schedules of the management data subscribed to by the manager
- the definitions of the notifications subscribed to by the manager
- the network-topology definition used by the network-map applet to construct its GUI
- the event-handler definitions
- the event-handler invocation log
- the pushed data
- a log of the pushed notifications
- a log of the events generated by the pushed-data interpreter
- statistical summaries of events

As we mentioned in the subscription phase, all these logically different data repositories may physically reside in one or more databases, NFS servers, DEN directories, etc.

6.3.5 Distribution

In NSM, scalability issues are typically addressed by distributing management over several machines. In Chapter 3, we identified three ways of delegating tasks in distributed NSM: by domain, by microtask, and by macrotask. Delegation by microtask is useful in manager-agent delegation, when the delegatee is only moderately smart (e.g., an SNMP agent); but it is not very interesting for manager-manager delegation, as managers are supposedly smart and can do much more than mere microtasks. In WIMA-push, we are therefore only interested in delegation by domain and delegation by macrotask. Let us describe these two approaches and how to combine them.

Management server viewed as a distributed system: delegation by macrotask

In the first scenario, the management server is viewed as a distributed system. The components described in Fig. 14 and Fig. 15 are now grouped into management *subsystems* and spread over different machines. We keep a centralized management paradigm, as we keep a single management domain for all the agents, but the management application is now distributed over several machines, each fulfilling a particular management task. In the case depicted in Fig. 16, mandatory management tasks are split into two subsystems: the event subsystem and the pushed-data subsystem. This way of distributing management tasks makes even more sense if we add the optional tasks sometimes found in management platforms (see Section 2.2); e.g., in Fig. 16, we show the billing subsystem; we could also add an accounting subsystem, a security subsystem, etc.

By balancing the load of the management server between several machines, we remain scalable up to a certain degree. But as the load caused by the management-application processing gradually increases, there comes a time when a single management task can no longer be executed on a single machine for all managed networks and systems (e.g., the rule-based event correlator of the event subsystem saturates and cannot keep up with all the events to process and rules to execute). At this stage, we need to go from one manager running on several machines (centralized management paradigm) to several managers running concurrently and in charge of separate management domains (weakly distributed hierarchical management paradigm).

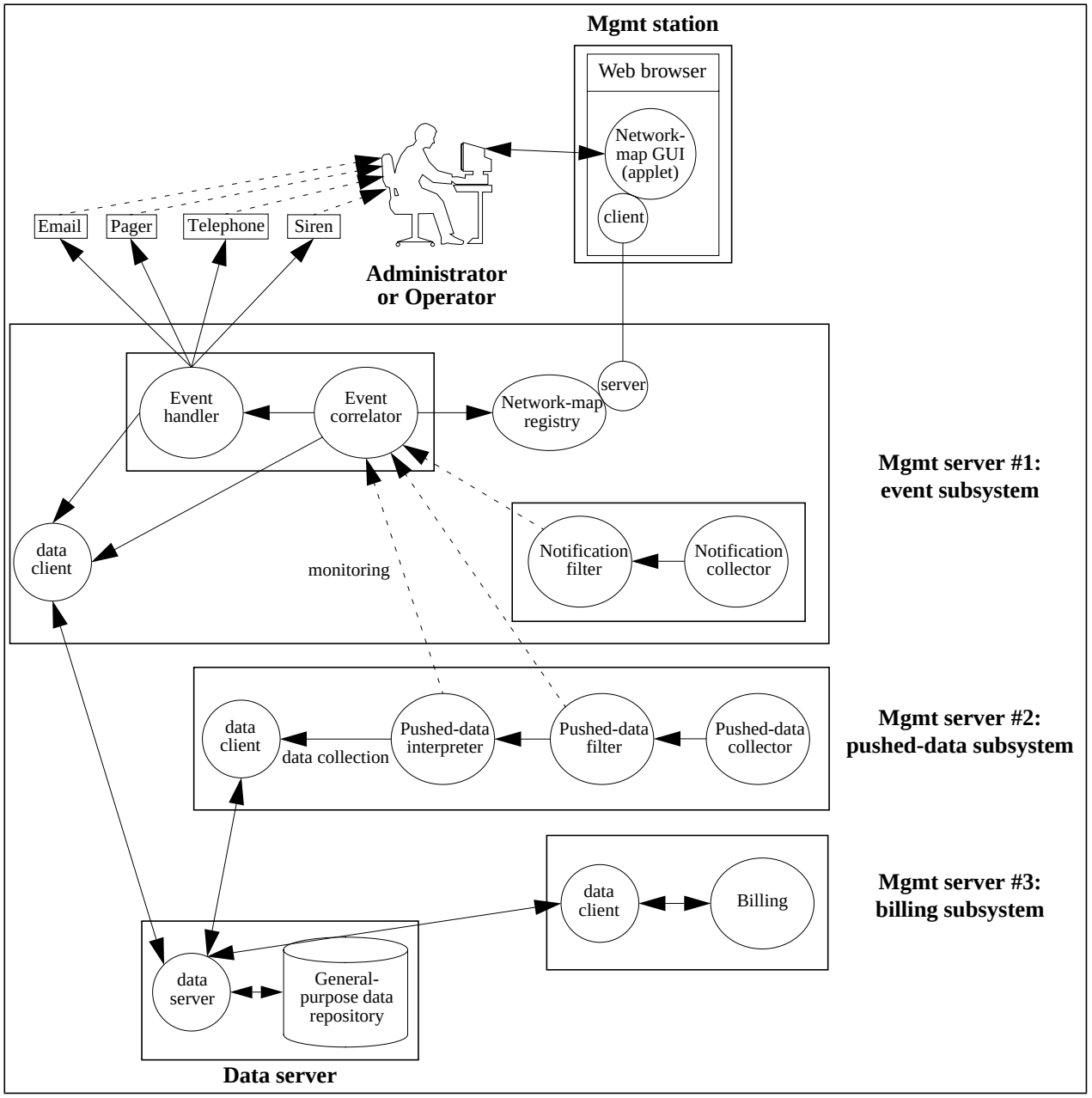

Fig. 16. Distribution: management server viewed as a distributed system

Distributed hierarchical management: delegation by domain

In the second scenario, we split one large management domain into several smaller management domains and assign one management server per domain. This solution presents several advantages. First, it enables administrators to decrease the amount of management data processed by each machine. As management domains can be further split recursively, this solution scales considerably better than the previous. Second, it is very well suited to geographically dispersed enterprises. By placing one management server per geographical location, the administrator saves a lot of bandwidth on WAN links, hence a lot of money. Also, for security reasons, it might simply be impossible for an agent to send data directly to a management server over a WAN link: the firewall protecting the management-server site might demand that this data be streamlined via a remote management server. Third, delegation by domain is a straightforward way to implement distributed hierarchical management (see our architectural decision in Section 6.1.4.4).

In the case illustrated by Fig. 18, we have one top-level manager, two management domains, and two mid-level managers. To keep this figure readable, we depicted only two agents, one in each management domain, but we would have many more in real life. A top-level manager looks very much like a management server in the manager-agent scenario studied in the previous sections. But a mid-level manager is somewhat more complex, as depicted by Fig. 17, because it glues together a management server (in charge of the agents located in its management domain) and an agent (which sends the data subscribed to by the top-level manager, as well as notifications).

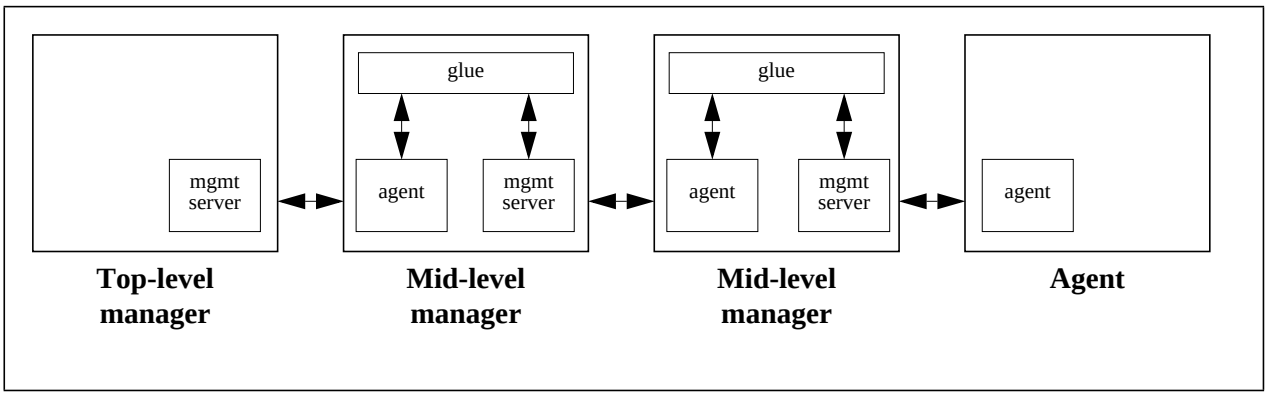

Fig. 17. Distributed hierarchical management: mid-level manager = management server + agent + glue

Fig. 18 shows that the event subsystem of a mid-level manager glues together the event subsystem of an agent and the event subsystem of a management server. Similarly, the pushed-data subsystem of a mid-level manager aggregates the pushed-data subsystem of an agent and that of a management server. Within a mid-level manager, some glue components allow the data received by the management server to be processed, and some results of this processing to be sent to the top-level manager. Inversely, other glue components allow high-level policies received from the top-level manager to be translated by a mid-level manager into low-level network and systems configuration data for the agents. The specification of these glue components is outside the scope of our Ph.D. work. It is complex and requires a well thought-out model to link policy-based management and NSM in both directions.

As we mentioned earlier, the organizational and communication models are identical between a top-level manager and a mid-level manager, between two layers of mid-level managers, or between a mid-level manager and an agent.

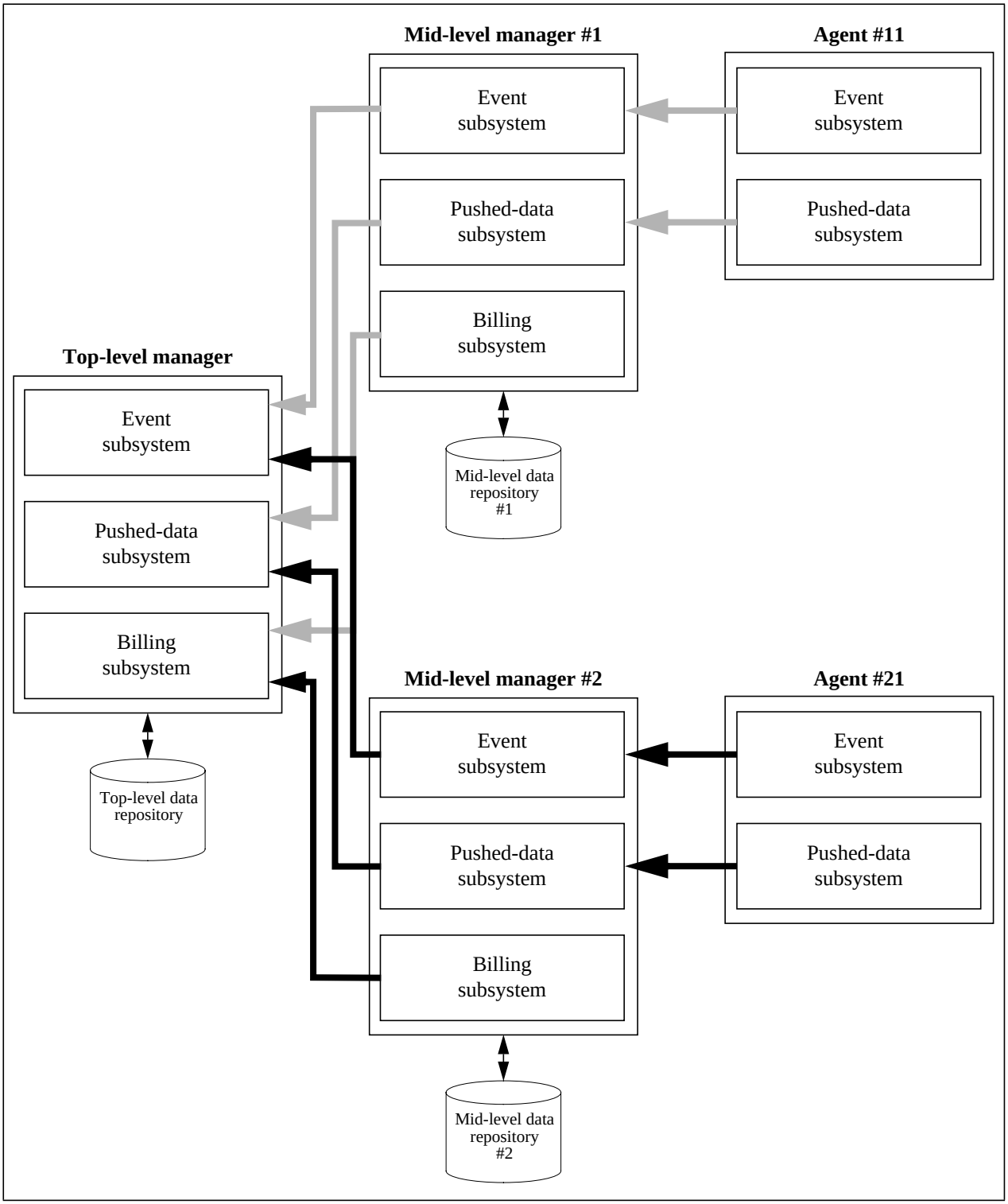

Fig. 18. Distributed hierarchical management: delegation by domain

Distributed hierarchical management: delegation by domain and by macrotask

The third scenario is a compound version of the previous two, whereby we integrate delegation by domain and delegation by macrotask. This scenario can be broken down into three variants.

In the first variant, we have strict delegation by domain and partial delegation by macrotask. For each management domain, the different subsystems depicted in Fig. 18 actually run on several machines (possibly one machine per subsystem). A typical case when this might be useful is when offline processing is very resource demanding over extended periods of time. As an administrator, you are interested in automating the generation of usage-statistics reports or per-department network-usage bills; but you do not want to slow down significantly the machine where the event correlator is running if offline processing is demanding in terms of CPU, memory, disk, and network bandwidth resources. By running the offline-processing subsystem on one machine and the event subsystem on another, we temporarily (or permanently) solve this problem, and we postpone (or avoid) the split of a management domain and the installation of a new manager, which saves work and money.

The second variant is the dual form of the previous: we have partial delegation by domain and strict delegation by macrotask. Within a management domain, all subsystems run on different machines. But some subsystems are delegated (e.g., the pushed-data subsystem) while others are not. In other words, only some management tasks are delegated by domain. For instance, billing can be entirely managed by the top-level manager, which directly accesses the data repositories of all the mid-level managers. Another example is event correlation: distributed event correlation is complex to model and understand; to avoid it, it might be interesting to only filter events at each mid-level manager, and to correlate events at the top-level manager.

In the third variant, we have partial delegation by domain and partial delegation by macrotask. This is more complex to do and more complex to debug than the previous variants. But it may be useful in particular situations, e.g. to cope with constraints imposed by legacy management applications. Although allowed in WIMA, this variant is not expected to be used in many cases.

We will come back to distribution in Chapter 8, when we explain how the use of XML can nicely unify and simplify the communication between managers.

6.3.6 Migration path: four-tier architecture

An important deployment issue is to define how to deal with legacy systems, namely SNMP agents (CIM agents are not deployed yet). How can we manage an agent that does not embed an HTTP server, or does not support the components described in Fig. 15 (push scheduler, data formatter, and data dispatcher)? In WIMA, we adopted a very simple solution inspired by the way SNMP agents were gradually deployed in the early 1990s: we add a management gateway between the management server and the agent. In other words, we go from a three-tier architecture to a four-tier architecture (see Fig. 19). The management gateway supports both push and pull. On the left side, it allows the management server to subscribe to management data, and it pushes regular data and notifications to the management server as if it were a full-blown WIMA agent. On the right side, it acts as a standard SNMP manager and polls the SNMP agent.

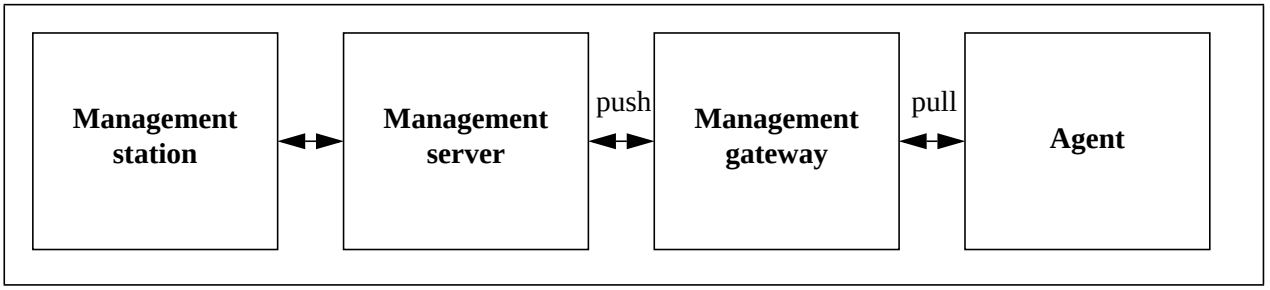

Fig. 19. Migration path: four-tier architecture

This simple migration scheme allows the agent vendor to integrate the management gateway inside the agent at any point in time (as we saw with SNMP in the early 1990s): the migration is then completed.

Note that having a single management gateway for several agents is feasible, but not very practical. We need to run multiple schedulers in parallel on the same machine, which is prone to interferences between the different emulated agents. But it is feasible and allowed by WIMA.

After having presented in detail the push model that underlies regular management and notification delivery in WIMA, let us now turn to the pull model.

6.4 *Ad Hoc* Management: The Pull Model

In this section, we present the details of WIMA-pull, our pull-based architecture used for *ad hoc* management. To begin with, we describe manager-agent interactions in two steps. In Section 6.4.1, we describe the simple case when we have no firewall and can access the agent directly from the management station (two-tier architecture); in Section 6.4.2, we explain how to deal with a firewall (three-tier architecture). Then, in Section 6.4.3, we describe manager-manager interactions and show how to distribute management. Finally, in Section 6.4.4, we present a migration path to deal with legacy systems.

6.4.1 Two-tier architecture (no firewall)

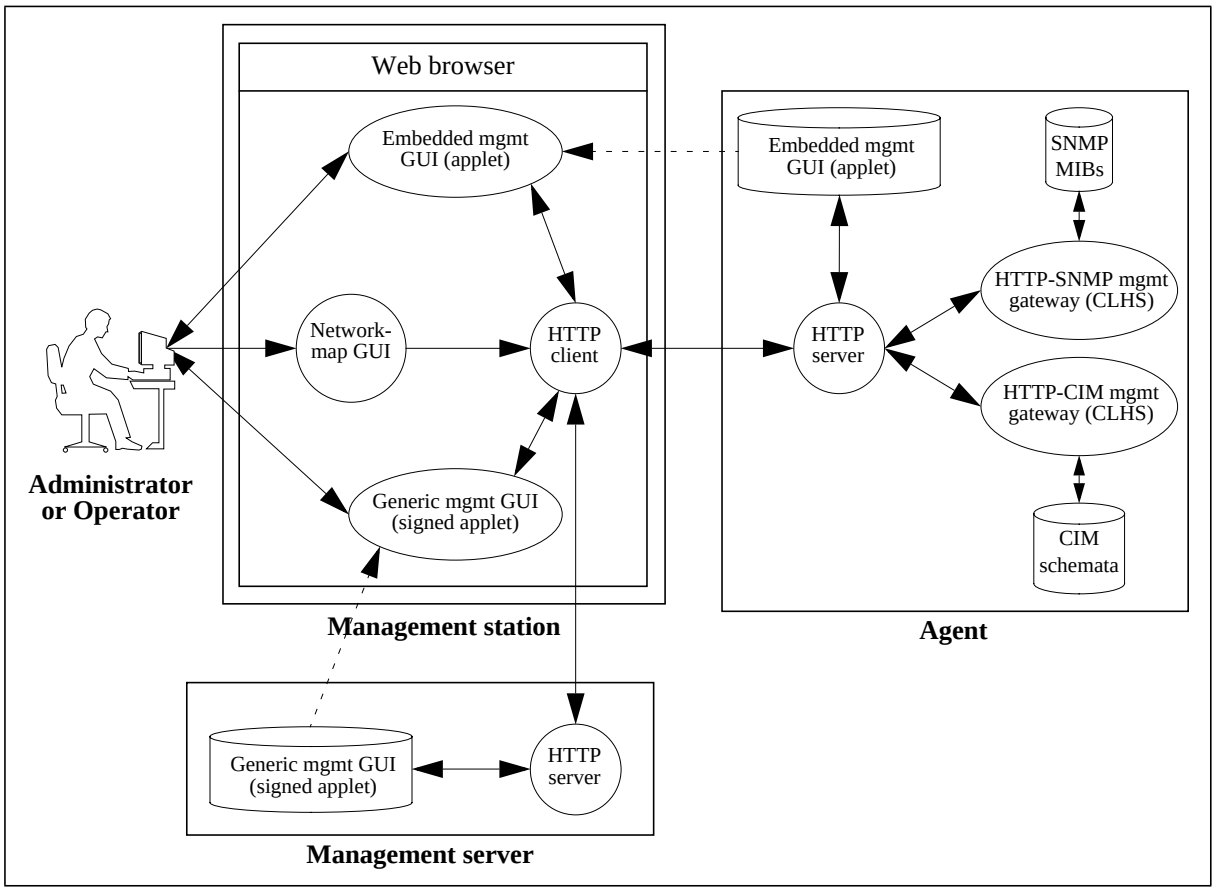

Fig. 20. Pull model: *ad hoc* management without firewall

Ad hoc management is typically used in interactive mode for troubleshooting or short-term monitoring (see Section 2.1.11). The management data retrieved from the agent is not stored or correlated on the manager side: it is immediately displayed (e.g., graphically as a time series, or textually as a text field) and discarded[1]. In WIMA, *ad hoc* management relies on the pull model for the reasons exposed in Section 6.1.4.3. Before we detail our general-purpose architecture that allows for firewalls between the agent and the management station, let us first study the simpler case when we have no firewall. Our three-tier architecture then becomes a two-tier architecture.

The direct interactions between the management station and the agent are depicted in Fig. 20. Similarly to the publication and subscription phases in the case of push, the starting point in the case of pull is the network-map GUI, a signed Java applet stored on the management server (see Appendix B). From this GUI, we retrieve the agent's management home page at `<http://agent.domain:280/mgmt/>` (see Section 6.3.3.2). From this home page, the administrator selects the URL of the embedded management GUI he/she is interested in (one agent may publish several embedded management GUIs on its home page). The management GUI, which is coded as a Java applet, is then loaded into the administrator's Web browser. Via an SNMP MIB browser or some kind of graphical tool, the administrator selects the SNMP MIB variables, CIM objects, etc. that he/she is interested in receiving. It is possible to tell the applet to retrieve the same entity every N seconds, and to abort the retrieval at any time. The details of the communication protocol will be described in Chapter 7.

When a MIB variable is requested by the applet, the request is made to the HTTP server run by the agent. This HTTP server then launches a CLHS that acts as a management gateway between HTTP and the virtual management-data repositories supported by the agent. In Fig. 20, we represented an HTTP-SNMP management gateway accessing SNMP MIBs and an HTTP-CIM management gateway accessing CIM schemata, but these examples are not limiting. Depending on the degree of optimization of the code run by the agent, the HTTP-SNMP management gateway can either directly access the in-memory data structures of the SNMP MIB (e.g., via shared memory) or do an explicit SNMP `get`. The same is true with other information models. This offers a useful migration path to network equipment vendors.

Note that the agent need not necessarily embed all the management GUIs that may be used by the administrator. It is also possible to retrieve a generic management GUI applet from the management server (this must be a signed applet, for the security reasons given in Appendix B in the case of the network-map GUI). Unlike embedded management GUIs, which are specific to a given agent, generic management GUIs are shared by many agents, possibly all of them. The main problem with generic management GUIs lies in the details of the communication protocol. In Chapter 7, we will propose conventions to exchange data between a management GUI (be it embedded or generic) and an agent. But the very nature of an embedded management GUI means that its vendor could adopt a proprietary communication protocol instead of ours to request an SNMP MIB variable, CIM object, etc. since the vendor codes both the embedded management applet *and* the components that underlie the management gateways running on the agent. If this is the case, the HTTP requests sent by a generic management GUI following our recommendations in Chapter 7 might not be understood by the proprietary management gateways embedded in the agent. There is a trade-off to be made: either the agent has the capacity to embed all the necessary management GUIs, in which case the vendor might be inclined to use a proprietary communication protocol, or the agent is very resource constrained and the vendor would better implement management gateways that support an open communication protocol.

Now that we have described a simple two-tier pull-based architecture that is only possible when we have no firewall between the agent and the management server, and is therefore not recommended in WIMA, let us present our complete, general-purpose, three-tier architecture that deals with (but does not require) firewalls for *ad hoc* management.

1. An entire time series may be saved in the general-purpose data repository, as a snapshot. Still, the main purpose of *ad hoc* management data is not to be stored and analyzed afterward, but rather to be analyzed immediately by a human.

6.4.2 Three-tier architecture (firewall)

As we explained in Section 6.2.2 (three-tier architecture) and Section 6.2.4 (firewalls), the management station should always access the agent via the management server. One of the reasons is that it facilitates access control and authentication. In Fig. 21, we added a third tier between the management station and the agent: the management server. This new tier is basically proxying all requests back and forth between the management station and the agent (e.g., for *ad hoc* monitoring), or between the management station and the data server (e.g., to save a time series for an SNMP MIB variable). This operation is performed by a component called the *secure relay*. Upon start-up, this secure relay retrieves the access-control lists and policies from the general-purpose data repository (usually, this sensitive data is not stored in the same database as the rest of the management data) and caches them locally. Once an administrator or operator has identified and authenticated himself/herself through a mechanism orthogonal to WIMA, this secure relay controls the access for that person on the agent and decides whether access to the agent should be granted to that person or to one of his/her requests. The access-control granularity, and more generally the security policy, is not specified by WIMA. It can be done on a per-agent basis, per-MIB basis, per-managed-object basis, etc.

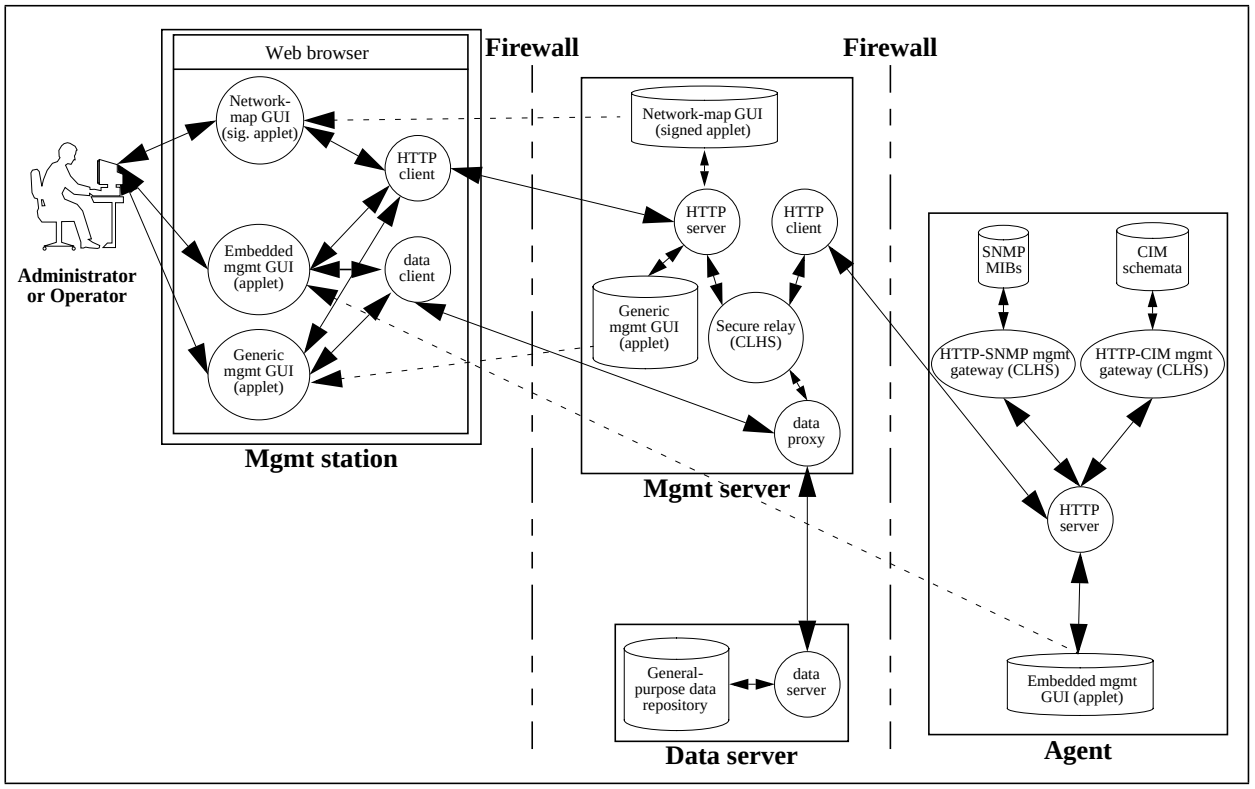

Fig. 21. Pull model: *ad hoc* management with firewall

Note that the component that implements this secure relay can be sub-divided into several components, and one of them can translate the communication protocol understood by the management gateways running in the agent into another protocol. This is particularly useful in the case of generic management gateways (see the problem exposed in the previous section). We will come back to this in Chapter 8, when we address the issue of translating between two information models.

6.4.3 Distribution

Distribution of management is generally useful when management is automated. It allows for the reduction of the number of events processed per management server, and generally speaking, mainly addresses scalability issues. *Ad hoc* management is always interactive, so it generally does not have to be distributed.

There is a noticeable exception to this: geographically dispersed enterprises protected by firewalls. For security reasons, the firewall of the headquarters (or the firewall of a subsidiary) might require that all the data coming in from a remote site (respectively coming out of this site) be streamlined via a single machine. In this case, an administrator located at the headquarters can temporarily monitor the error rate of a remote router at the subsidiary by going from his/her management station to the agent through the top-level manager (located at the headquarters) and a mid-level manager (located at the branch). Both of these managers can perform some identification, authentication, and access control. In this case, even mere *ad hoc* management requires a full-blown distributed solution. In the case of pull, as in the case of push, we use distributed hierarchical management in WIMA for the reasons exposed in Section 6.1.4.4.

6.4.4 Migration path

The migration path is trivial in the case of pull: the embedded HTTP server, the embedded *management gateway* components, and the embedded management GUIs described in Section 6.4.1 are initially externalized to a machine that acts as a standard SNMP manager toward the agent. Then, all of these entities can be integrated, agent by agent, as vendors add support for them. As in the case of push, we then gradually go from a four-tier architecture to a three-tier architecture (see Fig. 19), agent by agent, as the migration progresses.

Note that in the case of pull, a single external management gateway can be used for several agents. We saw the same thing in the early days of SNMP deployment, when management gateways supported SNMP on one side and proprietary management architecture and protocol on the other side. This is simple to do with SNMP because a single agent can support several MIBs, so one management gateway can support all the MIBs of all the agents. Similarly, in WIMA, one agent can manage multiple virtual management-data repositories (SNMP MIBs, CIM schemata, etc.), so it is fairly simple to group them logically and to make them all accessible from a single external management gateway.

6.5 Summary

In this chapter, we have presented one of the core contributions of this Ph.D. thesis: WIMA, our Web-based Integrated Management Architecture. First, we described our main architectural decisions. We explained why we focused on the definition of new organizational and communication models; we justified our choice not to define yet another information model; we highlighted the advantages of dissociating the communication and information models; and we proved that the push model is more appropriate than the pull model to transfer regular management data. Second, we detailed our main design decisions. We advocated the use of Web technologies, a three-tier architecture, and components; we took firewalls into account from the very beginning; we made the data repository independent of the management platform, and we made it easy to transfer management data in bulk; finally, we took several decisions that facilitate deployment, and explained how to deal with legacy systems. Third, we presented in detail our push model for regular management and notification delivery, and delineated event handling. Fourth, we described our pull model for *ad hoc* management. In both cases (push and pull), we showed how to distribute management across a hierarchy of managers and proposed a migration path from SNMP- to WIMA-based management.

Chapter 7

A NEW COMMUNICATION MODEL: WIMA-CM

In Chapter 6, we gave a broad view of how management data should be exchanged between a manager and an agent, or between two managers, using Web and push technologies. But what technology do the agent and the manager use to communicate? What protocol do they use to exchange management data? How should this data be formatted, represented, and encoded? What constraints must be satisfied when we have a firewall between the manager and the agent? In this chapter, we answer all these questions and define a new communication model called WIMA-CM (Web-based Integrated Management Architecture - Communication Model). This model consists of two parts. The communication model for push-based regular management is called *WIMA-CM-push*; its pull-based counterpart for *ad hoc* management is called *WIMA-CM-pull*. Most of our work went into defining WIMA-CM-push, which is one of the main contributions of our thesis.

This chapter is organized as follows. In Section 7.1, we express in simple terms the two communication problems implicit in Chapter 6. In Section 7.2, we describe our main design decisions to solve these problems. In Section 7.3, we highlight the drawbacks of using the sockets API as the manager-agent communication API. In Sections 7.4, we justify our choice for HTTP-based communication and describe WIMA-CM-push. In Section 7.5, we analyze the important issues of timeouts and reconnections. In Section 7.6, we present WIMA-CM-pull. Finally, we summarize this chapter in Section 7.7.

7.1 Two Communication Problems

In Chapter 6, we described a push-based management architecture for transfering regular management data and notifications (see Fig. 14 and Fig. 15), and a pull-based management architecture for transfering *ad hoc* management data (see Fig. 21). In this section, we extract two simple communication problems from these three complex figures: one for push, one for pull. The goal here is to identify the two ends of the communication pipe between the manager and the agent. These two simple models will be used throughout this chapter, when we describe the exact nature of the communication pipe and how to format the management data transferred across it.

7.1.1 Simplified terminology

To make figures easier to read and explanations less verbose in this chapter, the *management server* will be called the *manager* and the *managed entity* will be referred to as the *agent*. This is the case of terminology abuse mentioned in Section 2.1.4, whereby we confuse the management application with the machine itself. Moreover, each time we refer to manager-agent communication in a centralized scenario, we also implicitly refer to manager-manager communication in the context of distributed hierarchical management. In other words, whenever we use the word *agent*, we could also refer to another manager, one level down the hierarchical tree. The reason for this is that we use exactly the same communication model for manager-agent and manager-manager communication. We will come back to distributed hierarchical management in more detail in Chapter 8.

7.1.2 Communication problem for pushed data

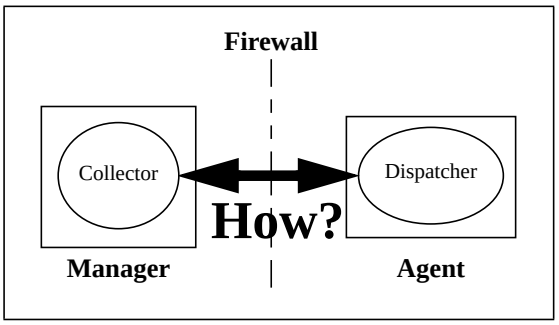

Fig. 22. Communication problem for pushed data

For the push model, whether we deal with monitoring, data collection, or notification delivery, we have a dispatcher on the agent and a collector on the manager, as depicted in Fig. 22. The dispatcher can be a pushed-data dispatcher, a notification dispatcher, or a combination of the two. Similarly, the collector can be a pushed-data collector, a notification collector, or both. As far as the communication between the manager and the agent is concerned, Fig. 14 and Fig. 15 can therefore be abstracted into a single communication problem shown in Fig. 22.

The dispatcher and the collector can be implemented as CLHSs (CGI scripts, CGI binaries, Java servlets, etc.) invoked via HTTP servers, or standalone programs (Java applications, C++ binaries, C binaries, etc.). They do not even have to be implemented with the same technology. In fact, the exact nature of the two ends of the communication path is transparent to the communication model itself.

7.1.3 Communication problem for pulled data

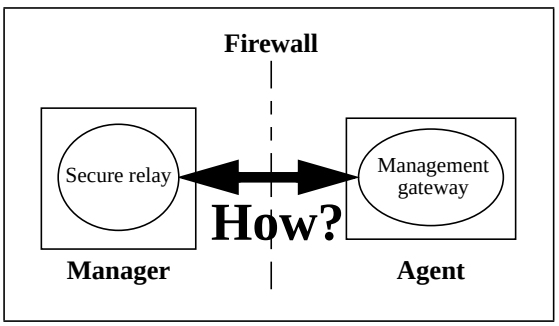

Fig. 23. Communication problem for pulled data

For the pull model, we used HTTP to communicate between the manager and the agent in Fig. 21, but we could have used a domain-specific transfer protocol over TCP or UDP as well. As a result, the HTTP-SNMP gateway depicted in Fig. 21 is replaced in Fig. 23 with a more generic management gateway, which may be internal or external to the agent. As in the previous case, the management gateway and the secure relay may be implemented as any kind of CLHS or standalone program. And once again, the technology used at the two ends of the communication path is transparent to the communication model itself.

7.2 Main Design Decisions

In this section, we present the main design decisions behind our new communication model: (i) the dissociation of the communication and information models; (ii) the use of a reliable transport protocol; (iii) the use of persistent TCP connections; (iv) the creation of the connections by the manager; and (v) the reversed client and server roles. When applicable, we point out the main differences between WIMA-CM and the communication model of the SNMP management architecture.

7.2.1 Dissociation of the communication and information models

As we saw in Section 6.1.3, one of our main architectural decisions was to dissociate completely the information and communication models in WIMA. The main characteristic of WIMA-CM, and in our view its main strength, is that it does not require or rely on the use of any specific information model (SNMP MIBs, CIM schemata, etc.). Instead, each bundle of management data is self-describing. The technology that we propose to use to implement this dissociation is MIME (Multipurpose Internet Mail Extensions). We will describe it in detail in Section 7.4.3, once we have justified a number of other design decisions.

7.2.2 Reliable transport protocol: TCP

In Section 2.4.1.8, we explained the problems induced by SNMP's use of an unreliable transport protocol. The simplest solution to these problems is to go from UDP to TCP to transport management data. This solution presents many advantages. First, it significantly reduces management-data losses. Urgent notifications sent by an agent are no longer lost for silly reasons such as buffer overflows in IP routers. We still have no guarantee of delivery, but at least the TCP part of the kernel tries hard (for 9 minutes in Berkeley-derived kernels) to send the data across. Second, it relieves the management application from managing all the timers and performing all the retries currently necessary to recover from network losses in the case of SNMP polling. Third, it reduces network latency. Fourth, it dramatically improves interoperability. Fifth, it makes it easier to go across firewalls. The first three advantages were already covered in Chapter 2; let us now expand on the other two.

7.2.2.1 Interoperability

Application-level acknowledgments (ACKs) are satisfactory in the case of SNMP polling because they are implicit and remain internal to the manager. If the agent does not answer to an SNMP `get`, the manager issues the same `get` again once a timer has expired. If the agent answers, it implicitly tells the manager that it has received its request, so the manager gets an implicit ACK from the agent.

The situation is very different in the case of push. The transfers of regular data are triggered by the agent, this time, not by the manager. If we use UDP to transport regular data, we need to define an explicit application-level ACKs policy for the manager to acknowledge receipt of the data. This requires that all agents and managers worldwide agree on the same ACKs policy and implement it in an interoperable way. How can we make Cisco, Nortel Networks, Microsoft, etc. agree whether ACKs could be piggy-backed or not, whether management data could be ACK'ed in bulk (i.e., with a single message) or not, whether the format of an ACK

could possibly or should necessarily include a Microsoft-specific OLE (Object Linking and Embedding) reference, etc.[1]? How can an ACKs policy be independent of the information model in an interoperable way? By using TCP instead of UDP, we free ourselves from defining such a scheme and we no longer have to convince all vendors to adopt the same ACKs policy. This in itself is a good reason to prefer TCP over UDP for push technologies.

7.2.2.2 Firewalls

Firewalls are easier to set up for TCP-based applications than for UDP-based applications [54, 55]. When a TCP client communicates with a TCP server, it is easy to learn who created the connection and who initiated a request; with TCP, it is difficult for an external machine to deceive an internal machine by sending a bogus response to a nonexistent request[2]. With UDP, the opposite is true: it is comparatively more difficult to learn who initiated a communication, but it is simplistic to send a response without having received a request first. The reason for this is that TCP is stateful whereas UDP is not. With TCP-based applications, TCP sequence numbers allow a firewall to easily match requests/responses coming from an external untrusted machine with the corresponding responses/requests issued by the trusted internal machine. Even better, by blocking incoming TCP segments with the SYN[3] bit set and the ACK bit clear, we prevent external machines from creating TCP connections to internal machines (we will come back to this in Section 7.2.4). Conversely, with UDP, we have no SYN bit and no sequence numbers. For security, UDP-based applications require smart and expensive *application-level gateways* (also called *UDP relays*) to provide for this functionality; such gateways render the two-way communication stateful and make sure that an external machine does not break the normal data flow of a UDP-based application.

So, TCP proves superior to UDP in two respects. First, TCP-based application-level gateways are easier to program than their UDP-based counterparts, because of the availability of sequence numbers in the transport protocol. Second, TCP-based applications can rely on simple *packet-filtering gateways* instead of full-blown application-level gateways, whereas UDP-based applications cannot; for UDP, application-level gateways are mandatory because packet-filtering gateways are too risky and renown for being inadequate—the main problem is the impossibility to prevent external machines from sending bogus responses [55]. This second argument is one of the main reasons why we selected TCP: packet-filtering gateways are considerably less expensive and less difficult to configure than application-level gateways. They can be integrated in IP routers, for instance. Although they are less secure (it is easy to make mistakes when defining the access-control lists of a packet-filtering gateway [55]), many enterprises are happy to use them. For SMEs, this is a very important argument. By selecting TCP, we do not require application-level gateways in our WIMA architecture whenever we go across a firewall: enterprises have the choice between inexpensive packet-filtering gateways and expensive application-level gateways. This is a significant improvement over SNMP-based management.

Another argument in favor of using TCP across a firewall is to use HTTP at the application level. The reason is that most firewalls worldwide are already configured to let HTTP traffic go through, because of the Web. The external Web server of the enterprise is usually not (and should not be, for security reasons such as denial-of-service attacks) the same machine as the manager depicted in Fig. 27. But because chances are that the firewall is already set up to let HTTP traffic go through in both directions, the update of the configuration file of the firewall is trivial: one only has to copy the configuration of the Web server and change the IP address of the machine allowed to go across the firewall. In many companies, especially SMEs, this does not require paying an (expensive) external consultant. We received very good feedback from the industry about this.

---

1. Note that SNMP-based management avoids this problem by not acknowledging notifications, which leads to some problems as we saw in Chapter 2.
2. It is not impossible, though: TCP sequence-number attacks make it possible to “steal” existing TCP connections. But they are only possible with poorly configured firewalls and with old TCP implementations that generate easy-to-predict sequence numbers.
3. SYN stands for SYNchronize. The SYN bit is one of the CODE BITS in the TCP header. It is characteristic of the first segment of a three-way handshake when a TCP connection is established.

7.2.2.3 Lightweight transport protocol

TCP is not an ideal transport protocol, though. It certainly relieves the application from performing retransmissions, but it also brings in a host of well-known problems [73]:

- TCP automatically comes with flow control, which may not be necessary. Flow control increases network overhead by dynamically adjusting the window size.
- TCP's slow-start mechanism is inefficient over fast network links, especially for LANs operating at 100 Mbit/s and above.
- TCP can be slow to recover from failures, particularly in case of heavy losses.
- TCP's urgent mode is messy and "leads to much confusion" [211, pp. 292–296]. Apart from that, TCP offers no way to differentiate between high- and low-priority segments within a single connection.
- TCP makes it impossible to time out a segment without timing out the entire connection.

If other lightweight, reliable transport protocols were widely available, they could be viable alternatives to TCP in NSM. Actually, this is not specific to NSM. A BOF (Birds Of a Feather) session chaired by Bradner and Paxson at the 43rd IETF Meeting [73] showed that many people working in different areas would be interested in an intermediary transport protocol halfway between UDP and TCP. This topic was also discussed within the IRTF Network Management Research Group, when we investigated whether administrators should be given the choice to transport SNMP over UDP, TCP [184], etc. The situation matured in the course of 1999, and the IETF Signaling Transport Working Group recently proposed the Stream Control Transmission Protocol (SCTP) [213]. Initially devised as a means to transport telephone signaling (e.g., Q.931 or SS7—Signaling System No. 7—) across IP networks, SCTP grew into a general-purpose transport protocol. After a long series of Internet-Drafts, SCTP should soon be submitted to the Internet Engineering Steering Group (IESG) for approval as a standards-track RFC. Whether SCTP will be accepted by the IESG and later by the market remains to be seen.

There are two well-known problems with defining new transport protocols. First, different applications have different requirements and want to change different things in TCP; this makes it difficult to find a consensus within the IETF around a single proposal. But most people agree that the market will be very reluctant to adopt several new transport protocols and will demand a single proposal. Hence, we face a deadlock. Second, there are so many TCP/IP stacks already deployed today that it would take many years and would cost a fortune to upgrade all of them to support the newly specified transport protocol. But nobody wants to use a new transport protocol that is not widely deployed, so we have a chicken-and-egg situation! One example is particularly symptomatic: even the designers of RTP (Real-time Transport Protocol) decided to layer it on top of existing transport protocols, although everyone agreed that a new transport protocol was needed for delivering real-time data.

If we remain pragmatic, we must therefore acknowledge that today, and in the foreseeable future, we have the choice between only two transport protocols: UDP and TCP. We explained why UDP is inadequate, so we have no other reasonable option than to adopt TCP.

7.2.3 Persistent TCP connections

Once we have selected TCP as the transport protocol, the next natural step is to use persistent TCP connections to exchange management data between the manager and the agents. The reasons are twofold. First, manager-agent associations are more or less permanent, or at least very long-lived: a given agent is managed by a given manager for extended periods of time. Second, persistency avoids the overhead of repeatedly setting up and tearing down TCP connections [200]; this problem is well-known and is the major reason why HTTP/1.1 is now gradually superseding HTTP/1.0. So, why do we not already use persistent TCP connections in SNMP-based management today?

7.2.3.1 The Myth of the Collapsing Manager

There is a well-established myth in the SNMP community that persistent TCP connections between the manager and the agents are not an option because of the memory overhead incurred by the manager when the number of agents grows large. After the *Myth of the Collapsing Backbone* and the *Myth of the Dumb Agent*, both destroyed by Wellens and Auerbach in 1996 [242], another SNMP myth is about to collapse: the *Myth of the Collapsing Manager*, or "Why persistent TCP connections are evil for transfering management data".

As a rule of thumb in production environments, it is often considered than one single manager should not directly manage more than a few hundred not-too-busy agents. Beyond 200–300 agents, we begin hitting well-known problems:

- The LAN segment to which the manager is connected becomes saturated (bottleneck effect).
- A single point of failure for so many machines and network devices becomes unreasonably risky. If one manager dies, too many agents are left on their own.
- The CPU and memory footprints of the management application become too large on the manager side: the event correlator cannot cope with the flow of incoming events, the pushed-data interpreter cannot cope with the number of rules to execute per push cycle, etc.

If the agents are particularly busy (e.g., if they often push data to the manager or if they often report problems), the administrator usually partitions the management domain and installs a new manager in the newly created domain. In such a setting, the maximum number of agents per manager falls well below the range given above, e.g., down to 50 (see Chapter 8 for distribution aspects).

We claim that the memory overhead of several hundred persistent TCP connections is perfectly acceptable on a modern management station. The reason why the *Myth of the Collapsing Manager* was born is that management stations had little memory when SNMPv1 was devised, typically 8–16 Mbytes. Management stations were typically workstations in those days, now they are PCs. Today, a basic desktop has 128 Mbytes of memory, and management stations typically have 512 Mbytes (sometimes more to manage large networks). This is more than enough to cope with hundreds of persistent TCP connections.

The Web gives us a solid argument to substantiate our claim. If we consider the constraints routinely satisfied by the Web servers of famous corporate organizations, it becomes obvious that our solution puts very reasonable requirements on managers. For instance, Kegel [119] reports that some sites routinely cope with thousands of concurrent HTTP or File Transfer Protocol (FTP) connections to a single host (not to a cluster), which is more demanding than our management scenario by an order of magnitude. Both Kegel and the `linux-kernel` mailing list [129] mention work under way to support more than 10,000 concurrent TCP connections, which makes our requirements comparatively minute (lower by two orders of magnitude).

Beyond these mere comparisons, can we put real figures on the memory overhead caused by hundreds of persistent TCP connections on the manager?

7.2.3.2 Assessing the memory overhead of persistent TCP connections is hard

Assessing the memory space required per open socket is a difficult task. First, it depends on the options that were selected when the kernel was built. In Linux, for instance, some C data-structure definitions contain optional fields declared between "`#ifdef`" and "`#endif`" block separators (e.g., the support for multiprocessor machines is optional). So, depending on the installation preferences specified by the administrator, the memory overhead caused by the creation of a socket may vary. Second, it depends on kernel caches, and the allocation scheme for new cache entries is operating-system specific. Third, the allocation of some data structures depends on the network topology because the manager caches the IP and Medium Access Control (MAC) addresses of its *neighbors* (see Fig. 24). By definition, a neighbor is adjacent to the manager; it is reached in one hop by a packet going from the manager to the agent (*destination host*). For instance, `router1` is the manager's neighbor for `agent1` in Fig. 24. Upon the creation of a socket, some data structures are allocated on a per-neighbor basis, others on a per-destination-host basis, yet others on a per-socket basis.

Clearly, the number of neighbors depends on the network topology, which is site specific and cannot be modeled *a priori*. Consequently, the memory footprint of the data structures allocated per neighbor cannot be calculated theoretically: we can only compute an upper bound for it.

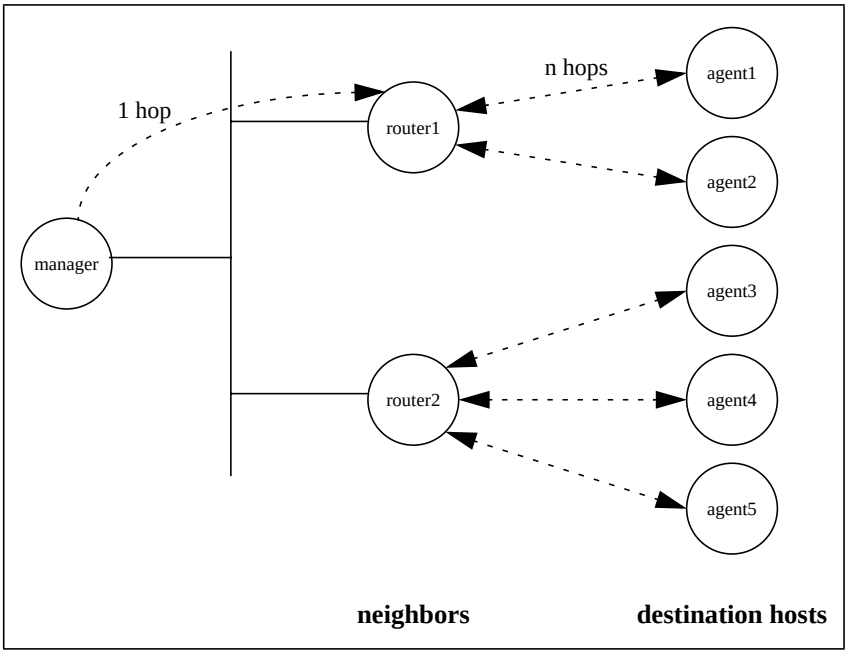

Fig. 24. Memory overhead of a socket: neighbors vs. destination hosts

Assessing the memory overhead of hundreds of active, concurrent TCP connections is even more difficult because of the sheer number of agents and the dependency on the network topology. First, experimentation is ruled out, because it is not feasible to gather hundreds of PCs in a test network, to use them to send real data to a manager, and to perform actual measurements on the manager. (We do not have hundreds of machines available for testing in our laboratory.) Second, it is not clear that results obtained from simulations would reflect the reality, as we will now explain.

7.2.3.3 Simulations: what about side-effects?

We just mentioned that data structures can be allocated per socket, per destination host, or per neighbor. As a result, the manager's memory overhead cannot be evaluated by simply creating hundreds of connections between two machines, one being the manager and the other simulating all the agents: we really have to simulate hundreds of *different* agents. To do so, we have two options.

In the first scenario, we preserve real manager-agent communication over a real network. For instance, we can simulate 100 agents on a single PC, run different copies of the simulator on several PCs sitting behind different neighbors, and have these PCs communicate with the manager over an Ethernet segment. On each PC, we can either impersonate 100 machines from a single kernel (e.g., by using virtual interfaces) or try to run 100 different kernels in user mode (quite a challenge but *a priori* not impossible).

In the second scenario, instead of having real communications over the network, we can instead have a single machine simulate all the agents, the manager, and all intermediate network equipment, interconnected according to a certain network topology. We can then simulate 401 kernels running in user mode as separate processes, separate threads, or a single process. Inter-machine communication is then replaced respectively with interprocess, interthread, or intraprocess communication. We also have to simulate by program the buffering and queuing mechanisms normally implemented by the kernels, as well as the network interfaces and the network links.

The problem with these scenarios is that they are exposed to side-effects which are difficult to assess. In the first scenario, would the simulator be able to deliver data via its single network interface in the same way that real-life agents would? No, because outgoing TCP segments supposedly generated by different machines would be queued in a single queue, outgoing IP packets supposedly coming from different machines would be queued in a single queue, data-link frames supposedly leaving through different network interfaces on different machines would be queued in a single outgoing queue and delivered to the network through a single network interface, etc. How far apart would simulation-based results be from real-life measurements? What would be the effects of this serialization? We do not believe that it is possible to know in advance. *A priori*, they could be close or remote as well.

In the second scenario, the connections created within a single machine (AF_LOCAL or AF_UNIX socket family) are different in nature and memory overhead from those created over a network (AF_INET socket family). Some network simulators, e.g. `ns-2` [228], do not even use sockets at all. So the measurements from simulations would necessarily be different from those measured in a real network. By which factor? Once again, it is very hard to tell. What about the side-effects of memory management? As we will see in the next section, this is already difficult to assess on a single machine running a single kernel; assessing it on a machine running 401 kernels in user mode seemed to be a daunting task to the author. For instance, if the kernel runs out of memory, allocations fail; but if user space runs out of memory, pages are swapped. Another source of side-effects is when the real kernel (the one running in kernel mode) runs out of free pages or buffers: all 401 simulated machines are affected, not just one. What impact does this have on the simulation results?

In short, if we had developed a simulator, we would not be able to justify why our results have some credibility. As a result, we opted instead for a theoretical study of memory overhead.

7.2.3.4 Theoretical study: *modus operandi*

Having ruled out experimentation and simulation, the only solution left to assess the memory overhead of hundreds of persistent TCP connections on the manager side is to perform a theoretical analysis. To do so, we[1] studied the kernel code (written in C) of Linux 2.3.99-pre6[2], the latest release of Linux at the time of writing. In order to compute this memory overhead, we added up:

- the memory allocated per socket;
- an upper bound for the memory allocated per neighbor;
- an upper bound for the memory allocated per TCP receive buffer; and
- a bit of extra memory as a rule of thumb to account for memory-management overhead.

Note that to assess the overall memory footprint, we should also account for application-level receive buffers (that is, the management-application variables where we store incoming data on the manager side). But these buffers are needed anyway, regardless of the transport protocol and the persistency of the connections. What we try to assess here is the extra memory overhead induced by the creation of hundreds of persistent TCP connections, not the overall memory footprint of the manager (which also runs the event correlator, the rule interpreter, etc.). Thus, we will ignore the application-level receive buffers in our calculations.

We decided to study not one but two cases. In Section 7.2.3.5, we investigate the worst-case scenario, which gives us an absolute upper bound for the memory footprint on the manager, whatever the network topology encountered, whatever the setup of the agents by the administrator. In Section 7.2.3.6, we present a more realistic scenario that takes into account some common-sense remarks. This allows us to compute a more realistic upper bound of the memory overhead.

In both cases, we assume that we have a maximum of 400 agents managed by a single manager, with two connections per agent: one for pushed data and one for notifications (we will justify these two connections in Section 7.4.2.2). This yields a total of 800 connections for the manager. The default number of file descriptors

1. The author thanks Werner Almesberger for his precious help in quantifying the memory footprint of TCP connections in Linux.
2. Linux 2.3.99-preXX are prereleases of Linux 2.4, expected to come out in the fall of 2000.

is large enough (4096 or 8192, depending on the options specified when the kernel was built, and dynamically settable via `/proc/sys/fs/file-max`), so we need not increase it on the manager.

7.2.3.5 Worst-case scenario

C data structures in Linux 2.3.99-pre6	bytes
`struct sock`	848
`struct inode`	404
`struct file`	96

Table 3. Memory overhead per socket

The footprint of the C data structures that are allocated whenever a socket is created is the following (see Table 3):

848 + 404 + 96 = 1348 bytes

With 400 agents and two connections per agent, this yields a total of:

1348 x 400 x 2 = 1078400 bytes (data structures allocated on a per-socket basis)

C data structures in Linux 2.3.99-pre6	bytes
`struct dst_entry`	100

Table 4. Memory overhead per destination host

The footprint of the C data structures that are allocated per destination host (i.e., whenever a connection is created to a new agent) is 100 bytes (see Table 4). We have two sockets but only one `struct dst_entry` per agent. With 400 agents, this yields a total of:

100 x 400 = 40000 bytes (data structures allocated on a per-destination-host basis)

C data structures in Linux 2.3.99-pre6	bytes
`struct neighbour`	112
`struct hh_cache`	44

Table 5. Memory overhead per neighbor

The footprint of the C data structures that are allocated per neighbor (i.e., whenever a connection is created through a new neighbor) is the following (see Table 5):

112 + 44 = 156 bytes

We already mentioned that the number of neighbors is site specific and cannot therefore be known in advance. Typically, it is very low (between 1 and 3) compared with the number of destination hosts (10s or 100s). In the worst-case scenario, we have one neighbor per destination host (agent). So, let us multiply the memory footprint of the C data structures by the maximum number of agents (400). This yields a total of:

156 x 400 = 62400 bytes (data structures allocated on a per-neighbor basis)

The total so far is thus:

1078400 + 40000 + 62400 = 1180800 bytes (total except TCP receive buffers)

C data structures in Linux 2.3.99-pre6	bytes
`(struct sock *)->rcvbuf`	max: 65535
`struct sk_buff`	164
`(struct sk_buff *)->truesize`	MTU + 4

Table 6. Memory overhead of TCP receive buffers

Let us now quantify the memory footprint of the buffers dynamically allocated by the kernel (see Table 6). We have one TCP receive buffer per connection. A TCP receive buffer is a virtual container of metadata and real data. In practice, it is a byte counter. The metadata consists in a doubly linked list of socket buffers (`struct sk_buff`) and a four-byte counter appended at the end of each data block (real data). A reference to this counter is stored in `(struct sk_buff *)->end`. The real data and the counter form a data block. A reference to the real data is stored in `(struct sk_buff *)->data`.

We have one socket buffer and one data block allocated per incoming packet (layer 3). The application-level data is chunked into a series of data blocks. An IP packet sent by the agent may result in more than one socket buffer allocated by the manager, because the IP routers between the agent and the manager may fragment the IP packets sent by the agent. If MTU discovery is enabled, it allows the agent to set the TCP's Maximum Segment Size (MSS) for that connection so as to avoid fragmentation. In this case, the number of IP packets sent by the agent is equal to the number of IP packets received by the manager. We need more than one socket buffer per connection on the manager if:

- the application cannot keep up with the amount of incoming data (CPU-bound);
- IP packets are delivered out of order; or
- IP packets sent by the agent have been fragmented.

Each socket buffer (metadata) has a memory footprint of 164 bytes with typical settings for building Linux 2.3.99-pre6. In the worst case scenario, the memory footprint of a data block is equal to the maximum valid MSS plus four bytes for the counter. The maximum valid MTU has a value of 65535 [148], which yields a maximum MSS of:

65535 - 40 = 65495 bytes

The maximum memory footprint of a data block is therefore:

65495 + 4 = 65499 bytes

On the manager, the actual size of a TCP receive buffer (`(struct sock *)->rcvbuf`) is set to `sysctl_rmem_default` by default (in `net/core/sock.c`). The default value of the latter is set to `SK_RMEM_MAX` in `net/core/sock.c`. The constant `SK_RMEM_MAX` is equal to 65535 (defined in `include/linux/skbuff.h`). Both the maximum and default sizes of a TCP receive buffer can be defined dynamically (that is, without rebooting the machine) by updating the following kernel pseudo-files:

```
/proc/sys/net/core/rmem_max
/proc/sys/net/core/rmem_default
```

The dynamically specified values of `rmem_max` and `rmem_default` cannot exceed `SK_RMEM_MAX`.

Both `rmem_max` and `rmem_default` apply to all processes running on the machine. But the size of a TCP receive buffer can also be specified by the application on a per-connection basis, by setting the socket option `SO_RCVBUF` (e.g., when the manager creates the persistent TCP connection). The buffer size specified with `SO_RCVBUF` cannot exceed `SK_RMEM_MAX`.

In the worst-case scenario, let us assume that the agents are pushing so much data so quickly that they fill up all the TCP receive buffers on the manager side. Although this worst case is not realistic, it gives us an upper

bound for the memory footprint on the manager. The receive window advertised by the manager ensures that the agent will not overflow the manager's TCP receive buffer. According to Stevens [212, pp. 191–192], this value used to be 4096 bytes in Berkeley-derived kernels, but is now typically somewhere between 8192 and 61440 bytes. This guarantees that the TCP receive buffer will not go beyond 65535 bytes for real data and metadata. In `net/core/sock.c:sock_rmalloc`, we see that Linux is even more flexible. The doubly linked list can grow as long as the aggregated size of the metadata and real data does not exceed the maximum allowed (65535 bytes). One last allocation can exceed this value, but all further requests for new allocations are refused until some data is read in by the application. If N_i is the number of bytes of real data received in the segment that fills up the i-th TCP receive buffer, the maximum memory footprint for the i-th TCP receive buffer is therefore:

$$(65535 - N_i) + 164 + N_i = 65699 \text{ bytes (independent of } N_i)$$

The total number of TCP receive buffers needed at one time on the manager depends on the number of agents pushing data at exactly the same time. This is site specific and varies over the time, so it cannot be assessed precisely. Let us be really pessimistic and assume that *all* agents are pushing data at the same time[1]. With 400 agents and one pushed-data connection per agent, this yields:

$$65699 \times 400 = 26279600 \text{ bytes (TCP receive buffers for pushed data)}$$

Notifications are short and fit into one TCP segment. They are seldom sent by agents. In a pessimistic scenario, let us assume that *all* agents use 1500 bytes per notification (Ethernet MTU) and have one socket buffer allocated on the manager side per notification connection. This yields:

$$(164 + 1500 + 4) \times 400 = 667200 \text{ bytes (TCP receive buffers for notifications)}$$

The total for TCP buffers is therefore:

$$26279600 + 667200 = 26946800 \text{ bytes (TCP receive buffers)}$$

This yields a total memory overhead of:

$$1180800 + 26946800 = 28127600 \text{ bytes (total without memory-management overhead)}$$

Finally, we need to account for Memory-Management Overhead (MMO). For instance, the amount of memory actually allocated in `net/core/skbuff.c` is rounded up to the closest higher multiple of 16 bytes. The worst case is encountered with the allocation of *slabs* (the memory-allocation units used for most dynamic memory allocations in Linux): whenever we call `kmalloc`, the amount of memory actually reserved by the kernel is rounded up to the closest higher power of two. So, if we allocate 300 bytes, 512 bytes are actually reserved. In the worst-case scenario, all dynamic memory allocations are of the form $2^N + 1$ bytes, where N is an integer, and result in actual reservations of 2^{N+1} bytes of memory. The worst MMO is therefore:

$$\frac{2^{N+1} - (2^N + 1)}{2^N}$$

This value tends asymptotically toward 100% when N grows. An upper bound for the MMO is therefore 100%. This yields a total memory overhead of:

$$28127600 \times 2 = 56255200 \text{ bytes} < 57 \text{ Mbytes}$$

In conclusion, the total amount of memory used by all the persistent TCP connections between the manager and the agents is always less than 57 Mbytes. Compared with the 512 Mbytes we expect the manager to have,

1. If this were the case in reality, it would probably mean that the administrator poorly configured the agents. It is simple for the manager to ask different agents to push regular data at slightly different times, either by explicitly requesting different absolute times, or by making sure that the agents use a different epoch t_0 to compute relative times. Agents may unpredictably change the time when they push data if their internal clock drifts, but the synchronization that we recommend in Section 10.4 makes sure that clocks do not drift excessively, which prevents accidental resynchronization of all push times.

this memory footprint is perfectly acceptable (11%). Even with only 256 Mbytes of memory, the manager would still cope with 400 persistent connections.

7.2.3.6 Realistic upper bound

In the previous section, we made several crude hypotheses that unduly increased the footprint of TCP receive buffers, and consequently vastly exaggerated the memory requirements on the manager. Let us try to define a more realistic upper bound of the manager's memory overhead by taking into account a few common-sense and practical remarks.

First, it is totally unrealistic to assume that the agents will fill up *all* of the manager's TCP receive buffers. In practice, a manager can be temporarily unable to cope with the amount of incoming data (that is, it is CPU-bound rather than I/O-bound). During a short period of time, some TCP receive buffers can indeed fill up, but only a fraction of them. If we choose a high upper bound, we can assume that up to 10% of the agents can fill up the manager's TCP receive buffers. Note that a manager should not remain CPU-bound over extended periods of time, say five or ten minutes in a row. If it does, it has a problem. Either its CPU is not powerful enough, in which case a CPU upgrade is needed, or its management domain is too large, in which case the domain should be split into two subdomains, with one manager per subdomain. There are cases, though, where none of these cures is possible—e.g., in case of budget restrictions. In this case, the administrator can prevent TCP receive buffers from clogging up the manager's memory by forcing its kernel to reduce the receive window of all the TCP persistent connections (`SO_RCVBUF` generic socket option). By doing so, we reduce the amount of data that the agents can push per connection, and we force the agents to buffer data in their TCP send buffers. The persistent connections are then said to be *receive-window-limited* [189].

Regarding the remaining 90% of agents, let us assume that the network is recent enough to support path MTU discovery [148] so as to avoid fragmentation (that is, IP routers support the *do not fragment* bit in the FLAGS field of the IP header), and that the receive window is small enough that we do not need many socket buffers to reorder the IP packets. In that case, we can suppose that the remaining 90% of agents require on average a maximum of three socket buffers per pushed-data connection at the manager (the value of three comes from Stevens [212, p. 192]), with an MTU equal on average to 1500 bytes (Ethernet MTU).

A second reason to reduce the memory requirements put on the manager is that notifications are rare events in the IP world. As a result, we should never receive non-stop streams of notifications from all agents at the same time, even in case of serious network conditions. We can therefore reasonably assume that only 1% of the agents have delivered notifications that could not be processed immediately. Notifications are short by nature and require only one socket buffer and one data block at the manager.

Third, based on typical LAN configurations, we might be tempted to assume that the number of neighbors rarely exceeds three, because a LAN segment is rarely interconnected by more than three IP routers. But with an intelligent hub, all the machines directly attached to the hub (respectively all the hosts belonging to the same virtual LAN) are (or appear to be) accessed in one hop. So, if the manager and the agents under its control are all connected to an intelligent hub (respectively belong to the same virtual LAN), all the agents will be both destination hosts and neighbors as far as the manager is concerned. The same thing is true if we replace the intelligent hub with switching equipment. In view of the generalized use of intelligent hubs in modern LANs, and in view of the current move toward switching equipment, let us assume that all destination hosts are also neighbors—that is, we have 400 neighbors (worst case).

Finally, the MMO was grossly overestimated in the previous section. If we consider the C data structures involved, we see that `struct sock` requires 1024 bytes instead of 848, which yields an MMO of 21%, not 100%. Similarly, `struct file` has an MMO of 33%, `struct inode` has an MMO of 27%, etc. All of these values are significantly lower than the MMO of 100% that we considered in the previous section. By looking at the different MMOs, we see that we can very safely assume that the MMO is 40%, as this value is still overestimated.

Under these new hypotheses, the overall memory footprint of the persistent TCP connections becomes:

1348 x 400 x 2 = 1078400 bytes (data structures allocated on a per-socket basis)
100 x 400 = 40000 bytes (data structures allocated on a per-destination-host basis)
156 x 400= 62400 bytes (data structures allocated on a per-neighbor basis)
(65699 x 400 x 10%) + ((164+1500+4) x 3 x 400 x 90%) = 4429400 bytes (TCP buffers for pushed data)
(164 + 1500 + 4) x 400 x 1% = 6672 bytes (TCP buffers for notifications)
1078400 + 40000 + 62400 + 4429400 + 6672 = 5616872 bytes (total without memory-mgmt overhead)
5616872 x 1.4 = 7863621 bytes < 8 Mbytes (total)

In conclusion, by making more realistic hypotheses, we have proved that in the worst case, we need less than 8 Mbytes of memory to cope with 400 agents. Compared with the 512 Mbytes we expect the manager to have in order to process the rules for so many agents, the amount of memory used up by all the persistent TCP connections between the manager and the agents is negligible (less than 2%). Even 128 Mbytes of memory would be sufficient. The management application itself, and most notably the rules used by the event correlator and pushed-data interpreter in large networks, require considerably more memory and render the issue of persistent TCP connections a nonissue in NSM. Hundreds of persistent TCP connections cause no significant scalability issue for modern managers. Another SNMP myth is dead.

7.2.4 Firewalls: persistent connections must be created by the manager

There is a principle of robustness in Internet security that says that TCP connections should preferably be created from a trusted host to an untrusted host, rather than the other way round [55, p. 56]. In our context, this means that the persistent TCP connections should be created by the manager (always trusted), not by the agent (possibly remote and therefore untrusted). By abiding to this rule, we protect ourselves against a number of well-known attacks.

The simplest example of such attacks is *intrusion* [54, p. 7]. If we allow certain external machines to connect via TCP to a reduced set of internal machines, e.g., with filtering rules on the access routers, the odds are that an attacker will find a hole in the filters and will manage to connect to an internal machine which was not properly protected against intrusions. One of the worst kinds of intrusion is `telnet`, which gives the attacker full access to a machine. There are ways to prevent such attacks, and most organizations are reasonably safe against intrusions. But the only way to be completely immune to this type of attack is to prevent external machines (e.g., agents) from connecting to internal machines (e.g., managers).

Another example of attack is *masquerade*, also known as *address spoofing* or *IP spoofing*. It is easy for an attacker to change its IP address to impersonate a remote agent and send management data on its behalf. The answers sent by the manager (e.g., the TCP ACKs) may not reach the attacker's machine, especially if the attacker operates in blind mode[1], but this does not prevent an attack from being conducted. For instance, an attacker can send bogus information to the manager, make it believe that the impersonated agent is experiencing problems, and entice a 24x7 operator to take drastic actions to try and recover from the bogus problem—thereby really breaking the network. By snowball effect, if many agents are impersonated and many actions are taken by the operators, the entire network of a large corporate organization or a large Internet Service Provider (ISP) can be brought to its knees. Panicking operators can take all sorts of decisions leading to all sorts of disasters—a human aspect often exploited by *social-engineering* attacks.

A third example is a class of attacks known as *Denial-of-Service (DoS) attacks*. First, an attacker typically probes all the TCP ports of the manager by trying to access all ports successively (there are ways to hide this kind of probing from simplistic pattern-matching firewalls). The purpose here is to learn what ports are active. Then, he/she connects to some or all active ports and swamps the manager with requests or data coming from

1. That is, the attacker does not control any WAN router along the path from the manager to the agent, he cannot eavesdrop traffic along this path, and he has no control over the agent. He can only keep the agent mute, e.g., by bombarding it with ICMP packets to prevent it from communicating with its WAN access router [55, p. 166].

many different machines (or even from a single machine if he/she is able to tamper with the IP address of outgoing packets). This is known as *flooding* [54, p. 8]. This can easily make the manager grind to a halt, thereby preventing 24x7 operators from doing their job and exposing the network to potentially serious problems. Instead of swamping the manager, the attacker can alternatively perform an attack known as *TCP SYN flooding*. By creating 10,000s of so-called *half-open connections* (a TCP connection torn down by its source immediately after the SYN packet of the three-way handshake is sent), the attacker can prevent agents from connecting to the manager by using up all available resources on the manager. Most TCP servers begin sending keepalives (`SO_KEEPALIVE` socket option) after two hours to clean up idle connections; so a manager may remain inactive for two hours after such an attack, a situation which may have a significant impact on a production environment. In case the TCP server is not configured to send keepalives (this is sometimes the case with old software), the manager remains inactive until it is rebooted!

By forcing the connection to be created by the manager, we do not prevent attacks altogether, but we make it harder for an attacker to succeed, especially if he operates in blind mode. By not being able to actively create new connections to the target of his attack, an attacker is forced to steal existing TCP connections. There are known ways of doing this (e.g. *sequence number* attacks allow for stealing existing TCP connections in blind mode [55, p. 24]), but they are much more sophisticated than the attacks we just described and require much smarter attackers.

Note that creating TCP persistent connections from the manager does not buy us total security. Some attacks are still possible, e.g., *sequence-number* attacks, replay, some DoS attacks, and attacks that require the attacker to read the packets sent by the manager to the agent. In environments that must be protected against such attacks, it is mandatory to use stronger security measures, e.g., Secure Sockets Layer (SSL [205]), IP Security (IPSec [209]) or data-link encryption [197]. The latter is the most expensive but can, under certain conditions [197], provide a very high level of security. But many sites do not want or cannot afford military- or bank-grade security, and are very happy with the sole protection offered by filtering gateways—which are typically set up to prevent the creation of TCP connections from the outside. Consequently, we need not (and do not) mandate strong security in our management architecture and communication model.

7.2.5 Reversed client and server roles

We just saw that, for security reasons, the persistent TCP connection must be initiated by the manager, not by the agent. Hence, when we go from the pull model that underlies SNMP-based management to the push model that we advocated in Chapter 6, the client and server roles are swapped. The transfer of management data is now initiated by the agent, instead of the manager; but the client side of the persistent connection (that is, the creator of the connection) remains on the manager, and the server side on the agent. Compared to the usual mapping between the manager-agent paradigm and the client-server architecture, the client and the server are on the wrong sides! Somehow, we want the server to initiate the communication, whereas the communication must be initiated by the client in a client-server architecture (see Fig. 25).

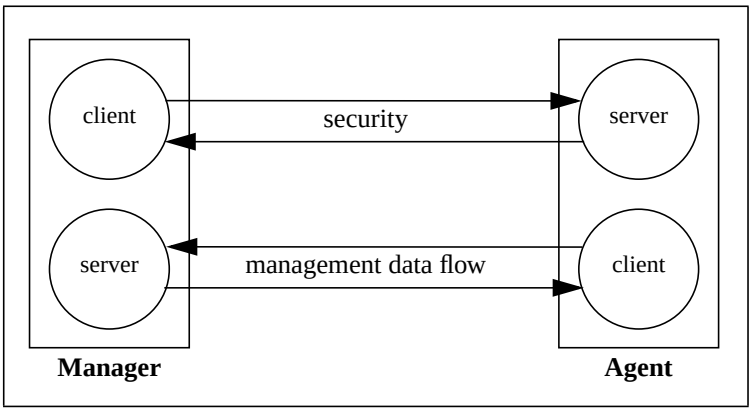

Fig. 25. Reversed client and server roles

To address this issue, we have four ways of communicating between the manager and the agent, that is, four APIs for writing the management application: the sockets API, HTTP, Java RMI, and CORBA [202]. We explained in Section 4.4 why Java RMI and CORBA are inadequate in the IP world (see the *my-middleware-is-better-than-yours* syndrome). So we have the choice between using HTTP as a high-level API, or plain sockets as a low-level API. The two solutions are not very different, because both solutions use a TCP connection for the communication pipe. But a few important discrepancies justify our preference for HTTP. We study sockets in Section 7.3 and HTTP in the remainder of the chapter.

7.3 The Sockets API as a Manager-Agent Communication API

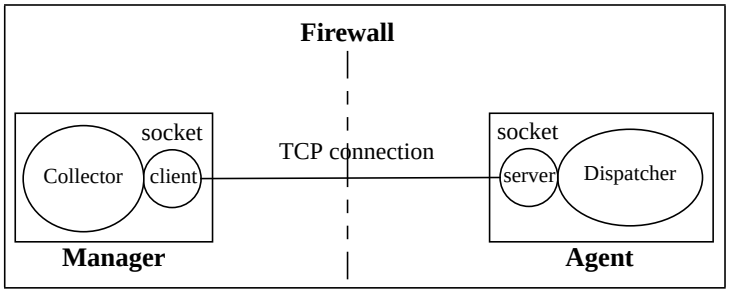

Fig. 26. The sockets API as a manager-agent communication API

Sockets are bidirectional. When a TCP connection is created, the TCP client contacts the TCP server, as usual in the client-server architecture. Once the connection is established, the client can send data to the server, but the server can also independently send data to the client via the same TCP connection. This property is very useful in our management scenario because it solves our problem of server-initiated communication. First, the manager creates a socket for each agent—that is, it creates virtual pipes to all the agents in its management domain. Later, the agents use these pipes to send management data across (regular data or notifications).

To ensure that the TCP connection remains persistent, the collector must not set a receive timeout on the socket when it creates it (`SO_RCVTIMEO`). If the underlying TCP connection is dropped for whatever reason (e.g., due to a network outage while the agent was pushing data), it is the responsibility of the manager to reconnect to the agent.

This solution offers a major advantage: simplicity. Socket programming is very easy in C, C++, Java, etc. But it also presents serious drawbacks. The main problem is the necessity to define a new, domain-specific transfer protocol *à la* SNMP. HTTP, built on top of TCP, is today's *de facto* standard in transfer protocols. Why should we use a domain-specific protocol rather than this standard protocol? The second problem is deployment. As we saw in Section 5.4, vendors now routinely embed HTTP servers in their network equipment, except in bottom-of-the-range equipment, and a large proportion of the deployed devices and systems already supports HTTP. As a result, management solutions based on HTTP can be easily deployed and adopted by the market. Conversely, deploying a new domain-specific transfer protocol built on top of the sockets API would take years, supposing that vendors all agree to adopt the same protocol. The third problem is related to firewalls. We face a potential problem if we need to go across a firewall between the manager and the agent. Most firewalls let only a few TCP ports go through [54]. So by default, firewalls will generally filter out the TCP connections that we intend to use between the manager and external agents. Thus, in order for this solution to work, all firewall systems must be modified. This may not be a problem for large organizations, because they generally have in-house expertise to set up UDP relays or update TCP filtering rules, or they can afford consultants to do the job. But it is often a problem for SMEs, who generally lack such expertise, and for whom expensive external consultants are only a last resort option. As we said in Section 7.2.2.2, this concern is not theoretical and was confirmed to us by people from the industry. The fourth problem is security. If we use the sockets API, we must use *ad hoc* solutions to secure the persistent TCP connection. Whereas by adopting HTTP, we automatically get several security mechanisms, e.g., HTTP authentication [80] and SSL [205].

Although these mechanisms do not offer strong security, they are better than the simplistic community string offered by SNMPv1 and SNMPv2c, and often meet the needs of customers who cannot afford expensive technologies such as data-link encryption.

For these four reasons, we prefer to base manager-agent communication (as well as manager-manager communication in a distributed scenario) on HTTP rather than on the plain sockets API with a proprietary transfer protocol and proprietary security mechanisms. But we stress that both solutions work.

7.4 HTTP-Based Communication for Pushed Data: WIMA-CM-push

HTTP does not exhibit the property of bidirectionality that we exploited previously. HTTP connections are oriented: it is not possible to create a persistent connection in one direction, from the client to the server, and later send data in the opposite direction. All HTTP methods rely on a strict request-response protocol; for an HTTP server to send a response to an HTTP client, it must have received a request from this client beforehand. It cannot send unsolicited messages.

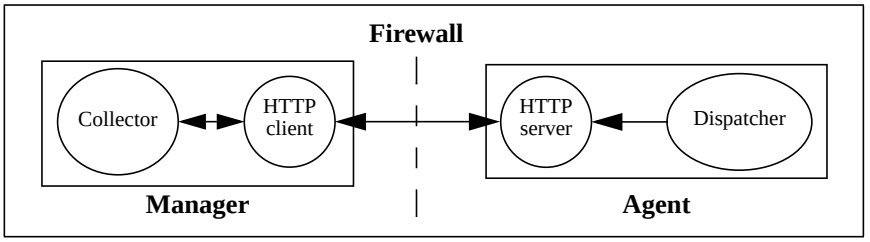

Fig. 27. Distribution via HTTP with a firewall

In this respect, SNMP and HTTP behave differently. SNMP follows a generalized client-server paradigm, whereby the response from the server can be either explicit (pull model of the `get`, `set`, and `inform` operations) or implicit (push model of the `snmpv2-trap` operation). HTTP, conversely, follows a strict client-server paradigm for all of its methods (`get`, `post`, `head`, etc.). With HTTP, the response from the server cannot be implicit. As a result, the implementation of push technologies with HTTP is not natural. How can we solve this problem?

7.4.1 MIME multipart

The first step toward a solution is to rephrase the problem as follows: How can we have an HTTP server send an infinitely large number of replies to a single request from an HTTP client? The second step is to make the agent send one infinite reply instead of an infinite number of finite replies. More specifically, we make the client send a single HTTP `get` request to the following well-known URL:

```
<http://agent.domain:280/mgmt/connect/all_data>
```

where `all_data` can be any kind of CLHS. In return, the server sends an endless reply which embeds separators in the payload of the HTTP message (see Fig. 28). To achieve this, Netscape proposed to use the `multipart` type of MIME [81, 153] in a different context (to update an applet GUI in a Web browser). We propose to use the same idea in NSM. At each push cycle, we send one MIME part including the description and value of all the MIB variables sent by the push scheduler. Once all the management data has been pushed, we send a MIME boundary; this metadata means *end of push cycle*. As to notifications, they are sent asynchronously, one at a time, through the same connection. In this case, the MIME boundary is interpreted as metadata meaning *end of notification*.

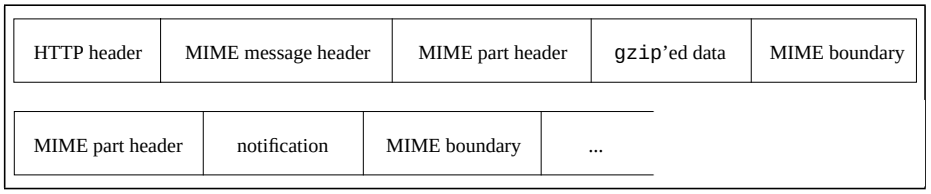

Fig. 28. TCP payload of the infinite HTTP response

HTTP/1.0 and HTTP/1.1 are not fully MIME-compliant protocols [74, Section 19.4.1]. But the use of the MIME `multipart` type is valid in both HTTP/1.0 [21, Section 3.6.2] and HTTP/1.1 [74, Section 3.7.2], so we have no problem here.

Note that we do not mandate the use of persistent HTTP/1.1 connections. WIMA-CM-push works with both HTTP/1.0 and HTTP/1.1. We did not want to rely on HTTP/1.1's persistence because the embedded HTTP servers that we find in network equipment today are often trimmed down versions of free HTTP/1.0 servers, with a low footprint and a well-proven record in terms of robustness.

Our solution presents two advantages. First, it is simple to implement. We will come back to this in Chapter 10, when we describe our prototype. Second, it makes it easy to go across firewalls, as we saw in Section 7.2.2.2. It is also exposed to two problems. One is the possibility to have an urgent notification significantly delayed by a large transfer of management data; this is addressed in Section 7.4.2, and leads us to slightly change this scheme. Another is the necessity for the manager to (i) detect when a connection is torn down and (ii) reconnect to the agent automatically; several solutions are described and discussed in Section 7.5.

7.4.2 Some notifications are more equal than others

In a LAN, we do not want to experience large transmission delays for urgent notifications. For instance, a high upper bound for push cycles (i.e., the maximum amount of management data pushed by an agent in a single cycle) is 50 Mbytes. This is transferred in less than 5 seconds over a 100 Mbit/s Ethernet segment, supposing that we have a large memory on the manager and a fast disk on the data repository. But we can experience long delays if we go across a WAN link. For instance, 1 Mbyte of management data takes 3 minutes to be transferred over a 56 kbit/s connection.

In most environments, urgent notifications should be processed within a few seconds, so the situation will be fine with most LANs but not acceptable with most WANs. The situation becomes worse in environments where urgent notifications ought to be processed within less than a second, typically when we have strong QoS requirements (e.g., network equipment sustaining IP telephony); in such cases, we have problems even with LANs. In other words, as our communication model should cope with all sorts of environments, we cannot wait for an entire push cycle to complete before we send an urgent notification.

We have two solutions to this problem. Either we interrupt the push cycle and send the notification immediately, via the same TCP connection, or we transmit the notification in parallel, via another TCP connection.

7.4.2.1 One TCP connection per agent: temporary interruption of the push cycle

The first solution assumes that we have only one persistent TCP connection per agent. In this case, we must interrupt the push cycle, send the urgent notification, and either resume the push cycle (send the remainder of the management data) or restart it (send all data again). We will face all sorts of problems if we adopt this solution.

First, how does the manager know that the push cycle has not cleanly completed but was aborted abruptly? As we do not have a separate control connection, we must embed a special *interruption string* in the payload of the HTTP message, e.g., the string "`<INTERRUPTED!!!>`", and make sure that this string never appears in

the original data. This can be achieved by (i) using some kind of domain-specific escape sequence, or (ii) using HTTP/1.1 chunked encoding, sending the interruption string as one chunk, and making sure that regular chunks do not consist solely of the interruption string. The former is not elegant in terms of design. The latter requires HTTP/1.1, which is incompatible with our desire to support both HTTP/1.0 and HTTP/1.1. In theory, an alternative to the interruption string is TCP's urgent mode, typically used to send the interrupt key in a `telnet` session. But Stevens recommends against using it because of the poor interoperability among different implementations [211, pp. 292–296], so we rule it out. In short, none of these solutions is really satisfactory.

Second, should the manager process on-the-fly the management data that it reads in, until a MIME boundary or the interruption string is encountered? Or should it buffer all incoming data until a MIME boundary is read in, and discard it without any processing if the interruption string is parsed? Clearly, parsing all the management data and buffering it before giving it to the pushed-data interpreter does not scale, so this solution should be avoided. Alternatively, we can buffer the events sent by the pushed-data interpreter to the event correlator. Once a MIME boundary is parsed, all buffered events are delivered to the event correlator; if the interruption string is encountered instead, the event buffer is emptied. The problem is that this scenario requires one event buffer per agent at the pushed-data interpreter, which does not scale. In short, buffer-based solutions are not adequate: the manager must process the data as it comes in.

Third, if the push cycle is interrupted to leave way to an urgent notification, the push cycle can either be resumed or restarted all over again. If we send the same data again with the same timestamp as before, the manager reads in the same data twice, up to the character where the previous interruption occurred. The manager therefore believes that the agent is bogus and sending the same data twice: it drops the connection to the agent and notifies the administrator that there is a problem with that agent (via the event correlator and an event handler). To allow the agent to resume the transfer, i.e., not to transfer again the data already received by the manager, we need application-level ACKs of management data. The problem with this approach is that it induces a very significant network overhead, which is precisely what we are trying to avoid (and one of the reasons why we decided to depart from the SNMP protocol in the first place).

In summary, none of these solutions is appropriate in the general case. Interrupting a push cycle to leave way for an urgent notification is not adequate.

7.4.2.2 Two TCP connections per agent

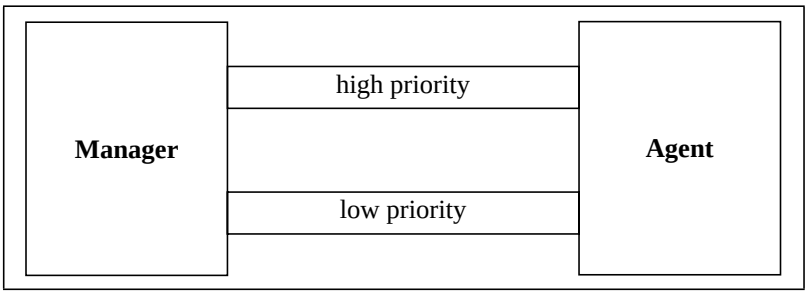

Fig. 29. Two connections per agent

The second solution is to transmit the notification in a separate TCP connection. This allows us to distinguish between high- and low-priority traffic (see Fig. 29). For each agent, one connection is dedicated to the delivery of urgent notifications, and another is used to transfer the rest of the management data (i.e., regular data and nonurgent notifications). The manager connects to the agent by requesting the following URLs:

<`http://agent.domain:280/mgmt/connect/high_priority`>

<`http://agent.domain:280/mgmt/connect/low_priority`>

where `high_priority` and `low_priority` can be any kind of CLHS. On both sides, the priority-based multiplexing between sockets can rely on multithreading or priority-tagged message queues.

Note that when we refer to *urgent* notifications, we do not have hard real-time constraints in mind, because they would require real-time operating systems on all the agents and the manager—which we usually do not have. The goal here is simply to avoid having to queue an urgent notification while Mbytes of nonurgent regular data are pushed by the agent. But it is no problem at all if a 64-kbyte TCP segment (the maximum for a single low-priority `write`) is entirely pushed by the agent via the low-priority connection before the priority-based multiplexing system switches to the high-priority `write`. Similarly, a low-priority notification can be entirely pushed via the low-priority connection before an urgent notification is pushed via the high-priority connection. The same is true for the manager, with low-priority `read`'s and high-priority `read`'s.

7.4.2.3 Generalization to multiple TCP connections per agent

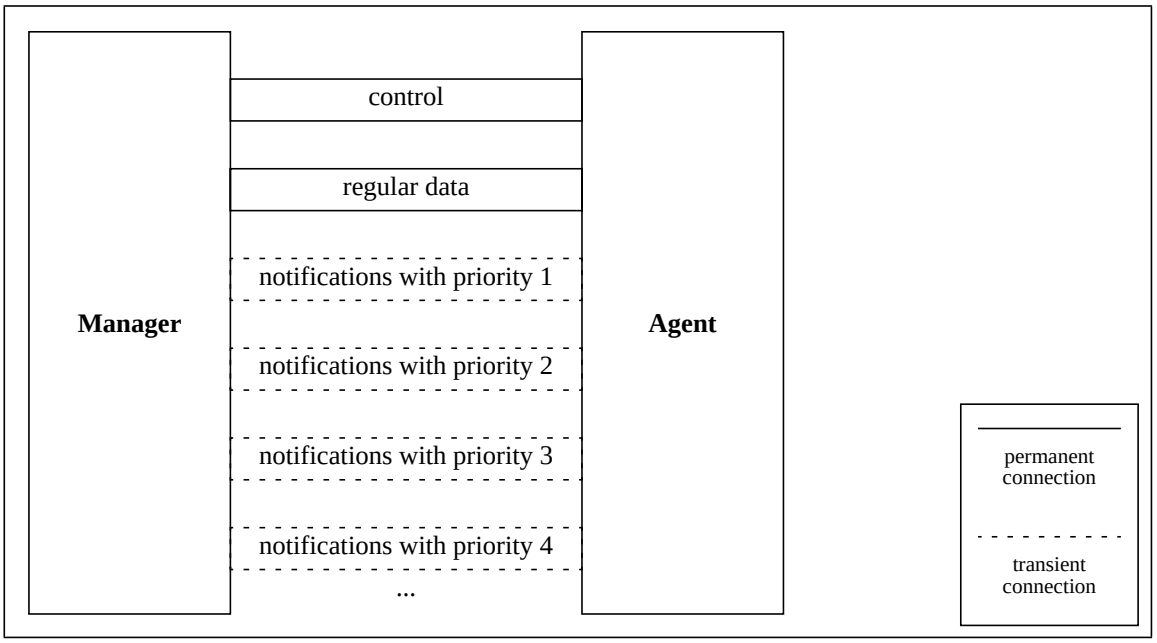

Fig. 30. Multiple connections per agent

In environments with complex QoS constraints, it may be necessary to distinguish more than two levels of priority[1]. For instance, in RFC 1470, there are four predefined levels of priority (called *levels of severity*): CRITICAL, ERROR, WARNING, and INFO. It would be a waste of resources to create one persistent TCP connection per priority level per agent. And in case the number of priority levels is not known in advance, it would not even be possible.

In such cases, we recommend to have one persistent control connection, one persistent connection for regular management data, and transient connections for notifications of different priority levels:

<`http://agent.domain:280/mgmt/connect/control`>

<`http://agent.domain:280/mgmt/connect/regular`>

<`http://agent.domain:280/mgmt/connect/transient?priority=string`>

Once again, `control`, `regular`, and `transient` can be any kind of CLHS. The control connection allows the agent to request new connections from the manager (*connections on demand*). When the control connection is created, and each time the manager creates a new transient connection to the agent, the manager sends a new HTTP `get` request to the agent via the control connection, issues a blocking `read` on the corresponding socket, and waits for a response from the agent; upon receipt of a response, the manager creates a new transient

1. The author thanks Thimios Panagos for suggesting this.

connection with the priority level specified by the agent, issues a new HTTP `GET` request, and enters the same loop. Per-socket timers on the manager make sure that idle transient connections are timed out after a while, e.g., 10 seconds, by using the `SO_RCVTIMEO` generic socket option. The full specification of the HTTP requests and responses is beyond the scope of this dissertation.

How does a manager find out whether an agent supports two or several levels of priority? The simplest solution for the manager is to request the two types of URLs, and depending on the errors that it gets from the agent, to opt for one solution or the other. Obviously, if an agent supports both schemes, the manager should select the most fine-grained scheme, that is, allow for multiple levels of priority.

7.4.3 Specifying the information model in the MIME header

Now that we have chosen to use MIME to separate different push cycles or different notifications, let us specify how we make management data self describing (see Section 7.2.1).

For describing the management data transferred in MIME parts, we use the *content type* of each MIME part. This description must include two things: the information model (SNMPv1, SNMPv2c, SNMPv3, CIM, OSI, etc.) and the encoding[1] (XML, string, BER, PER, serialized Java object, binary format, etc.). We have three levels of granularity for specifying the MIME type: the information model, the specification, and the mapping. The mapping is a set of rules for encoding a given information model.

Let us begin with examples of MIME types defined with an information-model granularity. If we assume that we encode management data in XML, we can define many MIME types for different information models, e.g.:

- `SNMPv1-to-XML`
- `SNMPv2c-to-XML`
- `SNMPv3-to-XML`
- `CIM2.2-to-XML`
- `CIM2.3-to-XML`
- `OSI-to-XML`

The problem with this approach is that it is too coarse grained. For instance, a certain SNMP protocol can be specified in several RFCs, with slight modifications due to the normal life cycle for an RFC to become a standard at the IETF. SNMPv3 has already been through three releases: RFCs 2261, 2271, and 2571. This can cause conformance and interoperability problems. A manager might claim to support a given information model, but slight differences between different versions of the specification could cause misunderstandings between agents and managers (see what happened with SNMPv2 in the mid-1990s). Thus, the exact version of the information model must appear in the MIME type. The same rationale is true for the encoding, although encodings normally change very little over time. We therefore need to go down to the specification version of both the information model and the encoding. In case the information model is SNMPv3 and the encoding is BER, we have three possible MIME types to date:

- `RFC2261-to-BER`
- `RFC2271-to-BER`
- `RFC2571-to-BER`

Unfortunately, MIME types are still ambiguous with this level of granularity. For given specifications of the information model and the encoding, we might have different versions of the mapping. For example, the DMTF released several versions of the CIM-to-XML mapping (or *xmlCIM* for short [67]); some even bear the same name but contain slight variations (e.g., xmlCIM 2.0 went through three releases in June 2, 1999, July 6, 1999, and July 20, 1999). The resulting ambiguity can lead to interoperability problems when an agent and a manager implement two different releases of the mapping. To avoid any compliance problem between two entities

1. Also called *representation*.

claiming to support the same information model and the same encoding, we need to specify the mapping version as well, as shown in the following examples:

- `CIM2.2-to-XML-v1.0`
- `CIM2.2-to-XML-v1.1`

At this stage, we have a very accurate way of specifying the information model, the encoding, and the mapping used by the agent. The problem is now the combinatory explosion of MIME types that must be created and registered with the Internet Assigned Numbers Authority (IANA [103]), each time a new version of an information model, an encoding, or a mapping is released. The poor scalability of this scheme makes it very impractical.

In order to reduce the number of MIME types required and the number of interactions with the IANA, we propose to define a single new MIME type (`application/mgmt`) and two MIME parameters, `mapping` and `version`, as shown in the following examples:

```
Content-Type="application/mgmt"; mapping="CIM2.2-to-XML"; version="1.0"
Content-Type="application/mgmt"; mapping="RFC2571-to-BER"; version="2"
Content-Type="application/mgmt"; mapping="CIM2.3-to-string"; version="3.0beta1"
```

This is the format that we mandate in WIMA-CM-push. We now have only one MIME type registered with the IANA. As far as the IANA is concerned, the names of the parameters are fixed, and their values are free-style strings. New valid values for the parameters can be defined over time, on an *ad hoc* basis, without contacting the IANA. Tomorrow, a new information model could appear, say XXX, which could be supported immediately with the following MIME-type parameters:

```
Content-Type="application/mgmt"; mapping="XXX-to-XML"; version="1.0beta1"
```

If we use the augmented Backus-Naur Form (BNF) defined in HTTP/1.1 [74, Section 2.1], the content type can be formally defined as shown in Fig. 31.

```
ContentType = “Content-Type=” NewMimeType Sep “ ” Mapping Sep “ ” Version
NewMimeType = <“> “application/mgmt” <”>
Sep = “;”
Mapping = “mapping=” <“> InformationModel “-to-” Encoding <”>
InformationModel = (*ALPHA Release)
Encoding = (*ALPHA Release)
Version = “version=” <“> Release <”>
Release = *(DIGIT | ALPHA | “.”)
```

Fig. 31. Formal definition of the content type

7.4.4 Optional compression of management data

Unlike SNMP, MIME supports a powerful feature: it distinguishes between *content type* and *content transfer encoding*. This allows for transparent compression of data in transit. Compression is optional in the case of MIME: some parts may be compressed, while others are not. This feature is useful in our NSM scenario in two respects.

First, agents that can compress data dynamically should preferably compress it so as to save network bandwidth, especially when regular management data is transferred in large bulks. But WIMA-CM-push does not mandate that all agents support compression. A direct consequence of this is that the manager should not necessarily expect MIME parts to be compressed. Within the same management domain, some agents may support compression and others may not.

Second, an agent may decide to compress some MIME parts but not all of them. For instance, notifications are inherently short; most of the time, they should not be compressed because the gain would be negligible, or could even be negative (e.g., `gzip` makes very small files bigger because it adds a header to the compressed data). Another example would be agents that compress data when their CPU load is low, but send data uncompressed when faced with a transient burst of activity.

Because of the expected heterogeneity of the agents, even within a single management domain, the manager should be able to process MIME parts that are compressed with different compression schemes (e.g., `gzip`, `bzip2`, `compress`, or `zip`), just like mailers do today. In other words, WIMA-CM-push does not impose any specific compression scheme. Typically, most if not all of the compression schemes listed by the IANA should be supported.

As we mentioned already, HTTP/1.0 and HTTP/1.1 are not fully MIME-compliant [74, p. 167], in spite of the repeated requests by the author to the IETF HTTP Working Group, in 1995, when the specification for HTTP/1.0 was written. As a result, there is no Content-Transfer-Encoding in HTTP. Fortunately, two HTTP header fields offer similar functionality: Content-Encoding and Transfer-Encoding. A Transfer-Encoding applies to an entire HTTP message, whereas a Content-Encoding applies to a single MIME part (*body part* in HTTP parlance). Clearly, we are interested in specifying the Content-Encoding header field in WIMA-CM-push. The valid values for the Content-Encoding (that is, the valid compression-scheme names in our case) are those allowed by the IANA [74, p. 23].

7.4.5 Example of HTTP and MIME-part headers

In Fig. 32, we give an example of HTTP and MIME-part headers for the infinite response sent by the agent.

```
HTTP/1.1 200 OK
Date: Wed, 10 May 2000 02:42:57 GMT
Server: Apache/1.2.4
Last-Modified: Thu, 23 Mar 2000 08:58:33 GMT
Mime-Version: 1.0
Content-Type: multipart/mixed;boundary=”RandomBoundaryString”

This is a multipart MIME message.

--RandomBoundaryString
Content-Type="application/mgmt"; mapping="CIM2.2-to-XML"; version="1.0"
Content-Encoding: gzip

Data for Part1

--RandomBoundaryString
Content-Type: "application/mgmt"; mapping="SNMPv2-Trap-to-BER"; version="2"

Data for Part2

--RandomBoundaryString--
```

Fig. 32. Push: HTTP and MIME-part headers of the agent’s reply

7.4.6 Simplifications in case we do not have a firewall

So far, we assumed that we could have one or several firewall(s) between the manager and the agent. But what happens if we know for sure that we have none? This case is of particular interest to ISPs, who control their entire network and usually do not need to access equipment via a firewall because their NOC is located at the

heart of their network. Let us first study what can be simplified in WIMA-CM-push under this new assumption, and then see if we can integrate the firewall and nonfirewall cases into a single communication model.

If we do not go across a firewall, we need no longer create the persistent HTTP connection from the manager. Instead, we can create it from the agent. This allows us to re-establish a normal client-server communication, and to put the HTTP client on the agent and the HTTP server on the manager. This new scenario is depicted in Fig. 33.

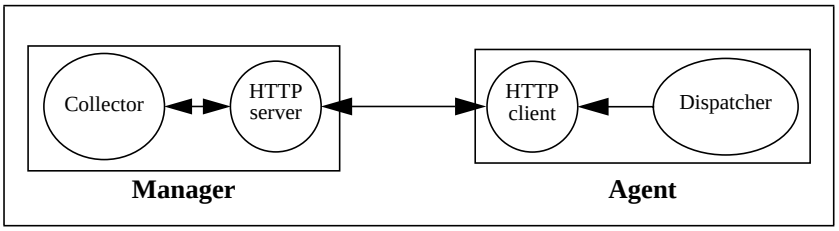

Fig. 33. Distribution via HTTP without firewall

This solution presents several advantages. First, it does not rely on nonintuitive designs that stretch the client-server architecture to its limits: the client is on the agent side and the server on the manager side. Second, the agent can reconnect immediately in case the persistent connection is broken: it does not have to rely on the manager to do that. This improves the robustness by avoiding time windows when the agent wants to send data to the manager but the manager has not yet reconnected to the agent. It also considerably simplifies the management of timeouts and reconnections (see Section 7.5), and suppresses the need for the manager to send keepalives. Finally, as in the firewall case, we do not need a specific version of HTTP: both HTTP/1.0 and HTTP/1.1 are appropriate.

The main drawback of this solution is that it breaks the nice unity of our management architecture. Firewalls, undeniably, bring in new constraints. But the advantage of using a single solution, whether we have a firewall or not, is homogeneity. An organization that has no requirement for a firewall today might have a very good reason tomorrow to put a firewall in place, e.g. if it changes its operation. An engineer working in a firewall-based organization might be transferred tomorrow to a firewall-free environment. An administrator coming from a small organization that could not afford a full-blown firewall based on application gateways could move tomorrow to an organization that can afford to buy a new firewall platform just to benefit from one single new feature. If we use the same management architecture everywhere, things become a lot easier to all these people. Of course, we could give the administrator the choice of creating the connections on the manager or agent side. But, by doing so, would we gain more than we would lose?

7.5 Timeouts and Reconnections

The issue of connection timeouts and reconnections is particularly important for two reasons. First, because of the reversed client and server roles (see Section 7.2.5), we must be careful that the manager detects a problem with the persistent TCP connection and reconnects to the agent in a timely manner. Otherwise, there will be time windows when the agent cannot send management data to the manager due to the absence of a persistent connection. This can cause robustness problems in the case of notifications, or buffering problems on the agent in the case of large push cycles. Second, the robustness of operating systems and Web-based applications relies partly on the automatic cleanup of broken TCP connections. If we do not clean up old broken connections, we might clog up certain resources on the manager and prevent new persistent connections from being created.

Persistent TCP connections can be timed out either by the operating system (transport layer) or the Web application (application layer). How do the timers in charge of this cleanup work? Are they compatible with our use of persistent TCP connections in WIMA-CM-push? What are the best strategies with respect to timeouts and reconnections? In Section 7.5.1, we investigate the timeouts performed by the agent’s and

manager's operating systems. In Section 7.5.2, we study the timeouts by the application, again on both sides of the communication pipe.

7.5.1 Timeouts by the operating systems

By default, idle TCP connections are not timed out by operating systems. This feature of TCP, often surprising at first, has some advantages (e.g., the network is not overloaded by heartbeat or polling overhead) and disadvantages (e.g., machines are exposed to DoS attacks such as *TCP SYN flooding*). By default, an operating system only cleans up a connection when it reboots or when an outgoing data transfer fails—typically in case of network outage, when one end sends data and the other end is unreachable. A direct consequence of this is that the sending end can time out a TCP connection but, by default, the receiving end does not.

In WIMA-CM-push, the management data always flows from the agent to the manager. The sending end is always the agent, and the receiving end is always the manager. So, by default, the agent can time out a connection but the manager cannot. This poses a problem, as it is much easier to control the behavior of one entity (the manager) than 10s or 100s of entities (the agents). In this section, we will see different ways of altering this default behavior of TCP.

An operating system can support two types of timeouts. Some apply to all the sockets managed by this operating system (per-kernel granularity), others to a single socket (per-socket granularity). Let us study these two types successively for the agent's and manager's operating systems (four cases in total).

7.5.1.1 Timeouts by the agent's operating system: per-kernel granularity

By default, the agent's operating system times out a persistent TCP connection by limiting the maximum number of retransmissions performed automatically by the TCP layer. The timer in charge of this is called the Agent's TCP Retransmission Timer (A-TRT) in WIMA-CM-push. This timer goes off after a certain time: the Agent's TCP Retransmission TimeOut (A-TRTO). A-TRTO can be expressed in seconds (it then takes continuous values on the time axis) or as a maximum number of retries (it then takes discrete values on the time axis). Let us assume that it is expressed in seconds.

In our management scenario, if the agent's operating system sends data across a persistent TCP connection and its TCP layer does not receive an ACK from the manager within a certain time (A-TRTO), then the agent's operating system drops the connection.

The behavior of A-TRT and the value of A-TRTO depend on the operating system. According to Stevens [211, p. 299], A-TRTO is hard-coded in the kernel of most Unix systems and cannot be modified. In Berkeley-derived implementations, for instance in 4.4BSD (Berkeley Software Distribution), it is equal to about nine minutes [211, p. 299]. This timeout value is the result of an exponential backoff with an upper limit of 64 seconds, which leads to the following series of backoff times between the 13 successive retransmissions:

$$\begin{aligned} \text{A-TRTO}_{\text{def}} &= \sim 1.5^{1} + 3 + 6 + 12 + 24 + 48 + 64 + 64 + 64 + 64 + 64 + 64 + 64 \text{ seconds} \\ &=\sim 542.5 \text{ seconds} \\ &=\sim 9 \text{ minutes } 2.5 \text{ seconds} \end{aligned}$$

The behavior of A-TRT is different in Linux 2.3.99-pre6. First, its backoff time does not have an upper limit of 64 seconds, but 120 seconds (the function `net/ipv4/tcp_timer.c:tcp_retransmit_timer` uses `TCP_RTO_MAX` which is defined in `include/net/tcp.h`). Second, the maximum number of retransmissions can be modified dynamically on a system-wide basis with the kernel pseudo-files `/proc/sys/net/ipv4/tcp_retries1` and `/proc/sys/net/ipv4/tcp_retries2`. Third, the successive backoff times are not statically computed as in 4.4BSD; instead, they are computed dynamically as a function of the Retransmission TimeOut (RTO), which itself is dynamically estimated with Karn's and

1. The first retransmission occurs after *about* 1.5 seconds, as explained by Stevens [211, pp. 236 and 299].

Jacobson's algorithms [31]. As a result, on Linux systems, A-TRTO can be larger or smaller than nine minutes. With the default values of `tcp_retries1` and `tcp_retries2`, it can be anywhere between 13 minutes and 38 minutes, depending on the estimated RTO (see `include/net/tcp.h`).

For the sake of simplicity, let us assume that A-TRTO is equal to nine minutes. This value is very reasonable in our management scenario, and it does not cause any problems in most environments. Let us also assume that the manager is not down for extended periods of time (that is, for more than A-TRTO seconds). The agent's operating system will time out the persistent TCP connection to the manager if, and only if, we experience a prolonged network outage (longer than A-TRTO) when the agent tries to push data. For instance, a network link might be broken, or an IP router along the path might be down. In this case, after nine minutes, the connection is closed by the agent but the manager does not know about it; we have a *half-open connection* clogging up the manager's memory. Worse, the agent knows that there is a problem, but cannot repair it for security reasons (as explained in Section 7.2.4); the manager could repair it, but it does not know that there is a problem. We have a deadlock! To break it, we must find a way for the manager to detect that the TCP connection was broken, so as to reconnect to the agent and restore the persistent communication pipe. (Remember that this persistent connection is relied upon by the agent when it sends management data to the manager.)

7.5.1.2 Timeouts by the manager's operating system: per-kernel granularity

TCP offers a simple way of making the receiving end detect that a connection was broken by the sending end: *keepalive probes*, or *keepalives* for short. Although not turned on by default, this feature is widely supported by current TCP/IP stacks. In this section, we study keepalives managed with a kernel-based granularity.

Keepalive probes: default settings

For the manager's operating system to become aware that the connection has been closed by the agent, the simplest is to send *keepalives probes* [211, pp. 331–337], a form of out-of-band data handled by the kernel and transparent to the application using the connection. To do so in our management scenario, the manager's collector must set the `SO_KEEPALIVE` generic socket option whenever it creates a persistent TCP connection to an agent [212, p. 185]. This socket option has been around for many years and is widely supported. On the manager, all the sockets that have this option set behave in the same manner. For kernels including a Berkeley-derived implementation of TCP (that is, most kernels available to date), this behavior is controlled by the following algorithm:

- As soon as the manager stops receiving data from the agent, it restarts its M-TKT (Manager's TCP Keepalive Timer).
- If a TCP connection remains idle for two hours, M-TKT goes off and the manager's kernel begins sending keepalive probes to the agent.
- As soon as the agent receives a keepalive probe, it must send an ACK to the manager.
- The manager's kernel waits 75 seconds between sending two consecutive keepalive probes.
- If the manager receives anything from the agent (ACK, data, etc.), it stops sending keepalive probes and restarts M-TKT.
- After nine successive keepalive probes are unsuccessfully sent by the manager to the agent, the connection is closed by the manager.

If the network outage ends while the manager is still sending keepalive probes to the agent, but after the agent has dropped the connection, the manager receives a reset (RST) from the agent. As a result, it drops the persistent TCP connection and reconnects to the agent. The chances for this reconnection to succeed are high, as the manager has just received some out-of-band data from the agent.

If the manager has not received any answer from the agent at the end of the keepalive-based probing, it drops the existing persistent TCP connection and attempts to create a new one to the agent. This time, the chances of success are slimmer, as the network outage could still be ongoing, or the agent might still be down.

The advantage of using keepalives is that it allows the manager to clean up sockets for broken connections, especially in case an agent reboots. This prevents broken connections from clogging up the manager's memory until the machine is rebooted. A minor disadvantage is that it requires to exchange a bit of extra out-of-band data between the manager and the agent, which slightly increases the network overhead; apart from bandwidth-starved WAN links, we can live with it happily in view of the offered functionality. A second drawback is that all sockets are treated alike; we will solve this problem in Section 7.5.1.4. A third issue is that the manager takes a very long time to reconnect to the agent (e.g., after the agent accidentally rebooted). The Manager's TCP Keepalive TimeOut (M-TKTO) is equal to:

$$\text{M-TKTO}_{\text{def}} = 7200 + (9 \times 75) = 7875 \text{ seconds} = 2 \text{ hours } 11 \text{ minutes } 15 \text{ seconds}$$

In other words, in case of prolonged network outage, it takes the manager more than two hours to discover that the persistent TCP connection was broken by the agent:

$$\text{M-TKTO}_{\text{def}} - \text{A-TRTO}_{\text{def}} = 7875 - \sim 542.5 =\sim 7332.5 \text{ seconds} =\sim 2 \text{ hours } 2 \text{ minutes } 12.5 \text{ seconds}$$

Somehow, this seems grossly inefficient! In most production networks, a management downtime of over two hours for all the agents in a given management domain is simply not acceptable.

Most modern operating systems solve or alleviate this problem by parameterizing some or all of the three values specified in the previous algorithm. These parameters, which we call the *keepalive-control kernel variables*, allow the administrator to customize the handling of keepalives for all the sockets of the machine that have `SO_KEEPALIVE` set. In Linux 2.3.99-pre6, these values are not only parameterized, they can also be updated dynamically (without rebooting the machine) via three kernel pseudo-files:

- `/proc/sys/net/ipv4/tcp_keepalive_time` controls the idle time after which the kernel begins sending keepalive probes.
- `/proc/sys/net/ipv4/tcp_keepalive_intvl` controls the time elapsed between two successive keepalive probes.
- `/proc/sys/net/ipv4/tcp_keepalive_probes` controls the number of keepalive probes that are sent before the kernel gives up and declares the other end unreachable.

The default values of these three keepalive-control kernel variables appear in Table 8. This applies to Linux. In perhaps the most famous textbook to date on the internals of TCP [249], Wright and Stevens describe the source code of another famous operating system, 4.4BSD-Lite, where the kernel variables have different names and slightly different semantics and default values. As this is confusing (and confused the author for some time!), the mapping between the two operating systems is given in Table 7:

Linux 2.3.99-pre6	4.4BSD-Lite
`tcp_keepalive_time`	`tcp_keepidle`
`tcp_keepalive_intvl`	`tcp_keepintvl`
`tcp_keepalive_probes`	(`tcp_maxidle`/`tcp_keepintvl`) + 1

Table 7. Mapping between Linux and 4.4BSD keepalive-control kernel variables

The main difference in the semantics is that in Linux, we count the first keepalive probe in `tcp_keepalive_probes`, whereas we do not in 4.4BSD-Lite's `TCPTV_KEEPCNT` (equal to the ratio `tcp_maxidle`/`tcp_keepintvl`). The default value of Linux's `tcp_keepalive_probes` is 9, while in 4.4BSD, `tcp_maxidle` is equal to 10 minutes, `tcp_keepintvl` is equal to 75 seconds, and `TCPTV_KEEPCNT` is equal to 8 [249, pp. 822–831]). Note that Stevens erred in [211, p. 332]: in 4.4BSD, a maximum of 9 probes (not 10 as stated) is sent, as explained by Wright and Stevens [249, pp. 830–831, Figs. 25.17 and 25.18].

In a companion book, Stevens also lists the keepalive-control kernel variables in Solaris, AIX, BSD/386, etc. [211, Appendix E]. They all follow 4.4BSD's naming convention, but some of these operating systems only allow one or two of these variables to be set. In our management scenario, the administrator should therefore be careful to select a modern operating system for the manager, in order to have good control over keepalives.

Keepalive probes: recommended settings

Now that we know how to modify these keepalive-control kernel variables, what values should we give them in NSM? How fast do we want to start sending keepalive probes? Should M-TKTO be larger or smaller than A-TRTO? The answers to these questions depend on the reactivity that the administrator expects from the manager, and this reactivity is highly site specific. If we assume that all agents take about nine minutes to time out a connection in case of network outage, we recommend that the keepalive-control kernel variables be assigned the values listed in Table 8. They yield the following M-TKTO:

$$\text{M-TKTO}_{rec} = 540 + (6 \times 10) = 600 \text{ seconds} = 10 \text{ minutes}$$

Linux 2.3.99-pre6 kernel variable	default value	recommended value in NSM
`tcp_keepalive_time`	7200 seconds	540 seconds
`tcp_keepalive_intvl`	75 seconds	10 seconds
`tcp_keepalive_probes`	9	6

Table 8. Per-kernel TCP keepalives: default and recommended values

As we can see, M-TKTO_{rec} is much more reasonable than M-TKTO_{def} in NSM. With our recommended settings, the manager can detect reasonably quickly that an agent has closed its side of the connection. A timeout of 10 minutes also corresponds to one of the typical polling periods[1] for a LAN in standard SNMP-based management, and a typical push period in Web-based management. Obviously, it makes sense to have an M-TKTO close to the push period. Moreover, by not being too close to nine minutes (that is, by selecting 600 seconds instead of 541), we allow (i) for the clocks of the manager and the agent to slightly drift apart, and (ii) for some kernel latency at the manager and the agent (e.g., in case several signals are queued). We also account for the imprecision in the value of A-TRTO_{def} (see footnote 1 p. 144).

An M-TKTO of 10 minutes might not be appropriate for all sites, though. For instance, it can be set to a much larger value (e.g., the default two hours) in a small LAN if reactivity is not of paramount importance and we simply want to clean up the manager's memory from time to time. By making M-TKTO bigger, we reduce the network overhead caused by out-of-band data.

M-TKTO can also be set to less than A-TRTO, say one minute. At first glance, such a setting might seem bizarre: we deliberately choose to send keepalive probes when the TCP connection is healthy, thereby increasing the network overhead. But there are cases where this behavior is desirable[2]. For instance, it allows the manager to detect a network fault before the agent; and the role of a manager is precisely to detect faults. It also enables the manager to detect the crash of critical agents in a timely manner; this might be very important in some environments. For example, if we want a critically important agent to send the manager a heartbeat every 30 seconds, M-TKTO should preferably be set to 35 seconds rather than 10 minutes. Because of the increased network overhead caused by reducing M-TKTO for *all* agents, there is a trade-off to be found between how quickly we want to detect that an important agent is down, and how much network overhead we are willing to pay for it.

1. The typical periods are 5, 10, and 15 minutes.
2. It may also cause problems. What happens if a manager reconnects to an agent before this agent has dropped the previous connection? We need a mechanism on the agent to drop the old connection and make the management application switch to the new connection.

Per-kernel timeouts: problems with heterogeneous agents

Per-kernel keepalives, and more generally per-kernel timeouts, work fine as long as we have a very homogeneous set of agents in a management domain. If all the agents have almost the same A-TRTO, it is fine for the manager to have the same M-TKTO for all its sockets. But the situation is very different when we have heterogeneous agents. Heterogeneous systems and devices have heterogeneous operating systems, hence possibly different kernel settings. A-TRTO may thus vary from agent to agent—e.g., we already showed that it can vary between Linux and 4.4BSD machines. As equipment vendors do not want to incur the risk that a piece of equipment stops working because an administrator mistakenly set some kernel variables to absurd values, it is generally not possible for administrators to configure A-TRTO in COTS agents. In practice, we must live with the fact that different agents have different A-TRTOs, and the best we can do is to try and address this heterogeneity problem on the manager side.

What value should we then assign to M-TKTO if most agents in a management domain have an A-TRTO of about nine minutes, but a couple of agents have an A-TRTO hard-coded to one hour? What happens if some agents do not support keepalives and therefore appear to be dead to the manager? How should we deal with agents that can unpredictably "freeze" their management layer over extended periods of time because other tasks have a higher priority[1]?

If we favor network overhead over robustness, M-TKTO ought to be equal to the largest A-TRTO of all the agents in the manager's management domain. A major problem with this approach is that modern, top-of-the-range IP routers might be unmanageable for two hours just because one old bottom-of-the-range device has an A-TRTO of two hours—this is simply not an option.

If we favor robustness over network overhead, M-TKTO should be equal to the smallest A-TRTO of all the agents in the manager's management domain. By doing so, we guarantee that problems with the most critical machines are detected as soon as possible. A major problem with this approach is that the manager keeps reconnecting to agents whose kernel does not support keepalives, or does not answer to keepalives for long periods of time, thinking that these agents have just rebooted. These spurious reconnections generate unnecessary network traffic, increase unnecessarily the CPU overhead of the manager and the agents involved, and cause many time windows during which the agents are not manageable, although in good health. A second problem is that if one agent has a really small A-TRTO, M-TKTO would also be unreasonably small, which would unduly increase the network overhead.

In short, choosing a system-wide, per-kernel keepalive policy to decide when a manager should reconnect to an agent is not easy in a heterogeneous environment. There is a trade-off to be found between robustness and network overhead, and the best value for M-TKTO is site specific. If we have a large distribution of A-TRTO values across a population of widely heterogeneous agents, there is simply no satisfactory M-TKTO.

Fortunately, all of these problems can be alleviated by controlling the manager's timeouts on a per-socket basis rather than on a per-kernel basis. Per-socket timeouts give us the granularity required in heterogeneous environments, hence in most real-life networks. Let us first study the agent side in Section 7.5.1.3, then the manager side in Section 7.5.1.4.

7.5.1.3 Timeouts by the agent's operating system: per-socket granularity

On the agent side, we have two means of overriding the per-kernel A-TRTO on a per-socket basis: the send timer and the retransmission timer. Both of these timers are managed by the kernel (kernel timers), but their timeout value can be altered by the application.

---

1. Several years ago, the author experienced such a problem with FDDI concentrators, whose management application was assigned a very low priority by the vendor in order to maximize real-data throughput. Their SNMP agents could remain silent to the manager's requests during several minutes in a row, up to half an hour in case of very large data transfers. This made the manager waste a lot of CPU cycles performing retries, and sometimes even convinced operators that a concentrator was down and should be rebooted.

The *send timer* is controlled by the SO_SNDTIMEO generic socket option (send timeout) [212, pp. 193–194]. It allows the agent's dispatcher to set a per-socket inactivity timer[1] to detect a network outage or a crash of the manager. With SO_SNDTIMEO, the send timeout can be different for different sockets on the same machine (unlike what we saw in Section 7.5.1.1). One advantage of this solution is that it allows the administrator to override A-TRTO when this per-kernel timeout value is not configurable. One problem with this solution is that Posix.1g does not mandate the support for SO_SNDTIMEO, although this socket option appeared with 4.3BSD Reno in 1990 [212, pp. 20,194]. As a result, some operating-systems vendors might not be inclined to support it. Linux added support for it very recently: SO_SNDTIMEO was not supported in Linux 2.3.28, released in November 1999, but it is supported in Linux 2.3.99-pre3, released in March 2000 (see net/core/sock.c:sock_setsockopt). To the best of our knowledge, it is not supported by commercial Linux vendors at the time we write these lines (May 2000). Therefore, COTS agents are likely to not support it (yet). We cannot rely on this solution in WIMA-CM-push.

The *retransmission timer* is controlled by the TCP_MAXRT TCP socket option (retransmission timeout) [212, p. 202]. It allows the administrator to override A-TRTO on a per-socket basis. TCP_MAXRT allows the agent's dispatcher to set a limit on the number of retransmissions performed by the agent's kernel if no ACK is received from the manager. TCP_MAXRT is specified in seconds by the application, but it can be rounded up by the kernel to the closest highest value accepted by the kernel [212, p. 202] (see the discrete values of the backoff times in Section 7.5.1.1). This rounding up is the main difference between TCP_MAXRT and SO_SNDTIMEO; apart from that, the end result is fairly similar. The main problem with TCP_MAXRT is that it is recent (it appeared with Posix.1g) and still rarely supported [212, p. 202]. It is symptomatic that the latest version of Linux, Linux 2.3.99-pre6 (released in April 2000), still does not support it (see net/ipv4/tcp.c:tcp_setsockopt), despite the well-known swiftness of the Linux community to implement new features. We therefore do not believe that TCP_MAXRT is a good candidate on the agent side.

The problem of course is whether equipment vendors will give administrators control over the embedded dispatcher. Unless the code of the dispatcher is downloaded into the agent dynamically or the dispatcher is written by a third-party vendor, the chances are slim. Although this solution works (e.g., SO_SNDTIMEO with Linux 2.99-pre6), we recommend working at the manager level instead.

7.5.1.4 Timeouts by the manager's operating system: per-socket granularity

On the manager side, we have two means of controlling connection timeouts on a per-socket basis: the receive timer and the keepalive timer. Both of these timers are managed by the kernel (kernel timers), but their timeout value can be altered by the application.

The *receive timer* is controlled by the SO_RCVTIMEO generic socket option (receive timeout) [212, pp. 193–194]. It allows the manager's collector to set a per-socket inactivity timer[1] to detect a network outage or a crash of the agent. With SO_RCVTIMEO, the receive timeout can be different for different sockets on the same machine (unlike what we saw in Section 7.5.1.2). One advantage of this solution is that the collector has to set the SO_RCVTIMEO option only once in the lifetime of the connection (we will see other solutions in Section 7.5.2 where this is not the case). Similar to what we said for SO_SNDTIMEO in the previous section, one problem with SO_RCVTIMEO is that Posix.1g does not mandate its support. But the main difference with the previous case is that we are now on the manager side, so the administrator can choose what operating system to run on this machine (this is not the case with most COTS agents, which generally come with an opaque operating system). The *receive timer* solution is therefore acceptable in WIMA-CM-push.

1. In Berkeley-derived implementations of TCP, the send timer is an inactivity timer, not an absolute send timer [212, p. 194]. It does not go off if a write takes longer than SO_SNDTIMEO seconds: it goes off if one TCP segment sent during a write operation is not ACK'ed by the agent within SO_SNDTIMEO seconds.

TCP socket option	Linux 2.3.99-pre6 kernel variable
TCP_KEEPIDLE	tcp_keepalive_time
TCP_KEEPINTVL	tcp_keepalive_intvl
TCP_KEEPCNT	tcp_keepalive_probes

Table 9. Keepalive control: mapping between TCP socket options and kernel variables

The *keepalive timer* is controlled by the TCP_KEEPALIVE TCP socket option [212, p. 201]. It allows the administrator to override M-TKTO on a per-socket basis. TCP_KEEPALIVE allows the manager's collector to set a limit on the time it takes the manager to time out a connection that does not answer to keepalive probes. In practice, this single TCP socket option is often replaced with three: TCP_KEEPIDLE, TCP_KEEPINTVL, and TCP_KEEPCNT, which correspond to the three keepalive-control kernel variables described in Section 7.5.1.2. This is the case in Linux 2.3.99-pre6 (see net/ipv4/tcp.c) and, according to the input we received on the USENET newsgroup comp.protocols.tcp-ip, also in 4.4BSD. In Linux, the mapping between these three TCP socket options and the keepalive-control kernel variables is straightforward, as depicted in Table 9. The four TCP socket options are linked by the following equation:

TCP_KEEPALIVE = TCP_KEEPIDLE + (TCP_KEEPINTVL x TCP_KEEPCNT)

In 4.4BSD and BSD derivatives, they are linked by the following equation:

TCP_KEEPALIVE = TCP_KEEPIDLE + (TCP_KEEPINTVL x (TCP_KEEPCNT + 1))

One problem with TCP_KEEPALIVE is that it is recent (it appeared with Posix.1g) and still rarely supported [212, p. 185]. However, TCP_KEEPIDLE, TCP_KEEPINTVL, and TCP_KEEPCNT were already supported in Linux 2.3.28.

The main advantage of the two solutions presented in this section is that they allow the administrator to modulate the reactivity of the manager depending on how critical a piece of equipment is to the robustness of the network. In Table 8, we already proposed some reasonable values in NSM for TCP_KEEPIDLE, TCP_KEEPINTVL, and TCP_KEEPCNT. But we saw in Section 7.5.1.2 that there are cases where we would like the timeout value on the manager side to be less than the timeout value on the agent side, and other cases were we would like it to be greater. For example, if we want a critically important agent to send the manager a heartbeat every 30 seconds, we can set the timer for that socket to go off after 35 seconds, not 10 minutes. Per-socket timeouts on the manager side give us the granularity necessary to manage heterogeneous agents.

A second important advantage of using per-socket timeouts on the manager side is that we can directly tie the receive timeout and the keepalive timeout values with the push period for each agent. This solution is very suitable in the NSM context. During the subscription phase, the manager must then register the smallest push period specified for a given agent, and later set the keepalive or receive timeout value for that socket to just above this smallest push period. This behavior is especially recommended for critically important management data, e.g. heartbeats.

The advantage of SO_RCVTIMEO over {TCP_KEEPIDLE, TCP_KEEPINTVL, and TCP_KEEPCNT} is that it does not cause extra network overhead. It does not require out-of-band data to be exchanged over the network. The disadvantage is that the possibility to manage per-socket keepalives is more widely available than the possibility to manage per-socket receive timers. But unlike what we saw with agents earlier, the administrator can choose what machine to use for the manager, so this last point is not really an issue. In conclusion, the best solution in our view is to select a manager whose operating system supports SO_RCVTIMEO, or otherwise to use {TCP_KEEPIDLE, TCP_KEEPINTVL, and TCP_KEEPCNT}.

To conclude Section 7.5.1, per-kernel timeouts work fine but do not offer us the granularity necessary in most real-life environments, whereas per-socket timeouts are easier to control on the manager side than on the agent side. So we should preferably use per-socket timeouts on the manager side. We have two options:

SO_RCVTIMEO and the trio {TCP_KEEPIDLE, TCP_KEEPINTVL, and TCP_KEEPCNT}. The former causes less network overhead, but the latter is more widely available. Both are appropriate in WIMA-CM-push.

7.5.2 Timeouts by the applications

In Section 7.5.1, we studied in detail what happens when the persistent TCP connection between the manager and the agent is dropped by the operating system at one of the two ends. Let us now investigate what happens when this connection is torn down at the application level. We distinguish four cases (see Fig. 27, p. 136). On the manager side, the connection can be broken by the HTTP client or the collector; on the agent side, by the HTTP server or the dispatcher.

7.5.2.1 Timeouts by the application on the manager side

Application-level keepalives are ruled out by the design decision made in Section 7.4.1. We send a single query from the manager and an infinite reply from the agent, so there is no possibility for the manager to regularly send application-level keepalives via the same connection. The only thing the manager can do is to restart a timer each time it receives data from the agent, and drop the connection and reconnect immediately when this timer goes off. We have three techniques to achieve this: SIGALRM, select and poll.

SIGALRM

The first technique relies on the SIGALRM signal. It can be broken down into three steps. First, the collector calls alarm and passes the timeout value as argument. This makes the kernel start the *alarm-clock timer*. Second, the collector issues a blocking read. Third, a SIGALRM signal is generated by the kernel when the timer goes off. As a result, the read system call is interrupted and the signal handler for SIGALRM is called instead. The problem with this solution is well known[1]: signals are not appropriate to handle traffic coming concurrently from many sources. First, we can have only one signal handler for SIGALRM per process or per thread. As the manager must communicate with many agents, we are forced to run many collectors (potentially up to several hundred) on the manager side, one per thread. Second, signal handling varies from machine to machine when Posix signals are not used, which makes the coding very difficult and error prone (Posix signals are not yet supported by all operating systems). Third, and worst, standard signals such as SIGALRM are not queued by the kernel and can be lost if several signals are generated in burst. This last problem renders this technique inadequate in our context.

select

The second technique is epitomized by the select system call in the C library, although it is not language specific. The collector issues a blocking select instead of a blocking read. With select, we can tell the kernel that we are interested in reading from several sockets, and how long we are prepared to wait. The timeout value is passed as argument to select; it controls the *select timer*. The select system call returns either when some traffic is coming from a socket or when the select timer goes off. We have two options: we can either run a single select and a single collector for all the sockets, or one select and one collector per socket (each collector then runs in a separate thread). With the first option, a single collector is connected to all the agents. This is very bad, because we then have a single timeout value for all the sockets (see the problems mentioned in Section 7.5.1.2). With the second option, we have one select timer per connection (hence per agent). If select returns due to a timeout, we drop the connection and immediately attempt to reconnect to the agent; otherwise, we read incoming data on the socket and re-enter the select loop. The advantage of select is that it is widely available. Unlike SO_RCVTIMEO and the keepalive-control socket options, which require recent operating systems, a timeout policy based on select can be implemented with virtually all

1. Stevens mentions this solution only in case we read from a single socket [212, pp. 349–352]. Kegel does not even mention it when he reviews I/O strategies for reading from many sockets [119].

operating systems. The main problem with `select` is that it is limited to FD_SETSIZE sockets; this limit is hard-coded in the kernel. FD_SETSIZE is often equal to 1024 [212, p. 152], but it can have a different value. With a maximum of 400 agents and two sockets per agent, we need 800 sockets on the manager, which is just below the maximum of 1024. This solution is therefore fine for WIMA-CM-push, but may cause problems in the future if, for some reason, we want to manage more than 400 agents per management domain. Other, rather technical problems with `select` are described by Banga *et al.* [17].

`poll`

The third technique uses `poll`, a variant of `select`. The internals of `poll` are different from those of `select`, but these two system calls offer similar functionality to the application programmer. They both take a timeout value as argument, they both return a readable socket or an error in case of timeout, etc. The main difference between these two system calls is that `poll` has no hard-coded equivalent to FD_SETSIZE. Based on this difference, some kernel experts claim that `poll` is superior to `select` for dealing with a large number of sockets [119, 212, p. 171]). Other kernel experts argue with this statement and claim that the implementation of `poll` is often inefficient, e.g. in Linux (see the `linux-kernel` mailing-list archive [129]); Banga *et al.* claim that `poll` is less efficient than `select` when the proportion of interesting file descriptors exceeds 3/64, because `select` only copies 3 bits per file descriptor while `poll` copies 64 [17]. When experts disagree, it usually means that there is no clear-cut winner. Another difference is that `poll` is less widely available than `select` [212, p. 172]. Note that a small variant of `poll` is to work directly with `/dev/poll` [119], which is faster than `poll` when the number of sockets is very large. As far as WIMA-CM-push is concerned, `select`, `poll`, and `/dev/poll` are all fairly similar. They all work fine in our context as long as we have only one connection per thread.

Signal-driven I/O and real-time signals

For the sake of completeness, we should also mention two other techniques: signal-driven I/O (Input/Output) and real-time signals. In case of signal-driven I/O, we tell the kernel to send a `SIGIO` signal to the application when something happens on a socket. Unfortunately, this technique is useless in the case of TCP sockets, as described in detail by Stevens [212, pp. 590–591]. In short, the problem is that `SIGIO` is generated too often by the kernel, and we do not precisely know what happened when its signal handler is called.

As for real-time signals, the problem is that they are rarely supported by current operating systems [119]. Unlike standard signals such as `SIGALRM` and `SIGIO`, real-time signals are queued and priority-based. As a result, we should never lose real-time signals unless the manager crashes (we can lose `SIGALRM` signals). Moreover, some sockets (hence some agents) can be assigned a higher priority than others, so management data coming from critical backbone routers can have precedence over data coming from mere hosts. These two properties are very useful in our management scenario. Unfortunately, real-time signals are still too rarely supported to be considered in WIMA-CM-push. Note that this may change in the future.

Collector vs. HTTP client

Now that we have investigated the different possibilities to time out the persistent TCP connection, let us see how to use them at the application level. On the manager, the TCP connection can be timed out either by the collector or the HTTP client. Presumably, the components used to implement the management application on the manager let the collector create the persistent TCP connection, and the HTTP client is just a library that offers a high-level API to perform an HTTP `get`, `post`, etc. If this is the case, all the system calls and signals studied previously apply to the collector. If, on the other hand, the HTTP client provides the collector with an opaque interface to the network and transparently handles all the reconnections and timeouts, then all the system calls and signals studied before apply to the HTTP client. The important point here is that it is transparent to WIMA-CM-push whether timeouts and reconnections are implemented by the collector or the HTTP client.

7.5.2.2 Timeouts by the application on the agent

On the agent side, there are two possible strategies to manage timeouts at the application level. One is to do this in the configuration file of the HTTP server; another is to configure the dispatcher. Both methods can be used concurrently, although only one is necessary.

Timeouts by the agent's HTTP server

For several years, developers of standard HTTP servers such as Apache [10] have protected their software against memory exhaustion caused by half-open connections. This problem, which used to plague busy HTTP servers on the Web, is often due to the impatience of users when heavy network congestion occurs. It can also be due to attacks, e.g. TCP SYN flooding, as mentioned in Section 7.2.4. Some users find it hard to wait more than a few seconds to download a Web page, and often reload the same page several times before it eventually gets completely transferred. By doing so, they create half-open connections on the machine hosting the HTTP server if the HTTP client does not cleanly close the TCP connection, or if the `close` does not reach the HTTP server due to heavy network congestion. For busy HTTP servers, half-open connections can very quickly use up all the memory available on the machine, thereby preventing the HTTP server from accepting new connections.

Two strategies have been typically used in the past to free the memory resources clogged up by idle connections. One is to assign a maximum lifetime to all HTTP/TCP connections and to time them out systematically after a certain time, regardless of whether they are idle or active. The other strategy is to assign a maximum period of inactivity to all HTTP/TCP connections, and to restart the inactivity timer of a connection each time some traffic goes across this connection.

In the days of HTTP/1.0, timeout values were often hard-coded in the HTTP server. With the advent of persistent connections in HTTP/1.1, the management of timeout values was refined. Most HTTP/1.1 servers today allow administrators to configure these timeout values from their configuration file. Some allow them to assign a maximum lifetime to all connections, others a maximum period of inactivity to all connections.

Note that there is a noticeable difference in scale between the Web and WIMA when it comes to timing out persistent TCP connections. On the Web, busy HTTP servers handle hundreds of thousands or millions of requests per day (e.g, for the Olympic Games [89]); in burst periods, this corresponds to tens or hundreds of requests per second. In NSM, supposing that the network is very unstable and that the manager keeps reconnecting to a critically important agent every minute (worst-case scenario), we still have only one new connection per minute. Moreover, persistent connections are short-lived on the Web; they are typically cleaned up after a maximum of 30 seconds, according to Gettys [102]. Conversely, in WIMA-CM-push, persistent connections are very long lived for regular management: we want the TCP connection between the manager and the agent to persist as long as both ends are up and running, hopefully for days or weeks. We want transient connections only in the case of *ad hoc* management (see WIMA-CM-pull in Section 7.6). So we do not want to set a maximum lifetime on incoming connections: the longer a persistent connection lasts, the better. We therefore recommend to clean up the agent's memory every two hours, that is, to assign a maximum inactivity period of 7200 seconds. Strictly speaking, we could optimize this value by making the inactivity timeout value slightly greater than the smallest period of all the push cycles registered with the agent, and updating it whenever the manager changes its subscription. But in practice, the extra functionality does not justify the extra code that must be embedded in the agent to achieve this.

One may argue that cleaning up the agent's memory so rarely makes it vulnerable to DoS attacks based on half-open connections. But the best way to shield an agent from DoS attacks is clearly not to ask it to time out idle HTTP connections very often. It is the task of a firewall to protect internal machines from external attackers; agents have other tasks to do. Without firewalls, DoS attacks can fill up the memory of all critical agents in a matter of seconds. This is independent of WIMA, and even independent of Web-based management.

Timeouts by the agent's dispatcher

If we have little or no control over the configuration of the agent's HTTP server (e.g., if we use an HTTP server with a minimum footprint such as Hong's Embedded Web Server [115]), an alternative is to control the timeout in the dispatcher. But we already mentioned that access to the internals of COTS agents is very unlikely in practice. Unless the dispatcher is downloaded by the administrator into the agent (mobile code), it is preferable to control timeouts in the HTTP server's configuration file.

7.5.3 Synthesis

To conclude with timeouts and reconnections, let us summarize our recommendations and explain what to do when a manager fails to reconnect to an agent. We will also address a special case that we purposefully skipped thus far to simplify the organization of this long section.

Recommended solutions

Per-kernel timeouts are easier to set up than per-socket timeouts, but they do not offer the granularity that we seek (in real-life networks, some agents are more critical than others), and they are inappropriate in the case of heterogeneous environments. So we recommend to use per-socket timeouts.

On the agent side, we should rely on the default settings of the kernel and application because most COTS agents do not let the administrator control their internals (retransmission timer, send timer, etc.). The control of timeouts and reconnections should therefore be achieved by the manager. In most environments (especially those with no hard real-time constraints), we do not need to control timeouts on both sides of the communication pipe simultaneously.

On the manager side, timeouts can be managed at the kernel or application level. Four solutions are appropriate in our context: `SO_RCVTIMEO`, the trio {`TCP_KEEPIDLE`, `TCP_KEEPINTVL`, and `TCP_KEEPCNT`}, `select`, and `poll` (with its variant `/dev/poll`). The first two are slightly better because they need to be installed only once (`select` and `poll` need to be restarted each time we have processed an incoming `read`). One issue is that three out of the four solutions require a modern operating system on the manager. In some environments, this is not a problem because the administrator can control what machine to use for the manager; in this case, we recommend using `SO_RCVTIMEO` or `/dev/poll`. In other environments, this may cause some problems; we then recommend using `select` as long as the expected number of agents within a single management domain does not exceed 400.

Note that different solutions can be combined. For instance, if we do not have a multithreaded operating system on the manager, we can use `SO_RCVTIMEO` to specify per-socket timeouts, and we can block in read on all sockets simultaneously with `select`, `poll`, or `/dev/poll`.

Note that there is no need to impose a specific way of timing out idle connections in WIMA-CM-push. This is specific to a given manager and independent of the agents. Within a hierarchy of managers, some managers may use one scheme while others use another. Our recommendations here are destined for developers of component-based management applications. The main goal of this section was to prove the feasibility of our solution (our design decisions do not lead to a dead end), identify the issues, and assess the trade-offs.

Informing operators

Regardless of the technique actually used by the manager to reconnect to the agent, it is important to note that machines (agents) sometimes remain down for long periods of time in real-life environments, possibly several days. It would therefore be very inefficient for the manager to retry indefinitely to reconnect to an agent until this agent eventually comes back to life. A better scenario is to make the collector send an informative alarm

to the event correlator as soon as a persistent TCP connection is timed out, then to retry to reconnect twice (which yields a total of three attempts[1]), and in case of failure, to give up and send a critical alarm to the event correlator. The corresponding event handlers could make the corresponding icon on the network-map GUI turn yellow and red, respectively, for instance. It should be possible for operators to resume the management of an agent from the network-map GUI, e.g., through a context-sensitive pop-up menu. Resuming the management of this agent would involve creating a new persistent TCP connection from the manager to the agent, which would trigger the push scheduler on the agent side.

Note that the collector should also inform the pushed-data interpreter that the agent is considered to be permanently down and is therefore no longer managed. By doing so, the collector informs the pushed-data collector that rules related to that agent should not be executed until further notice.

Write timeouts by the manager

There is one rare case that we did not investigate thus far: write timeouts by the manager (we only studied write timeouts by the agent). The only time the HTTP client can time out a connection while performing a `write` is immediately after the creation of the TCP connection, when the HTTP client sends its single HTTP `get` request. If the agent crashes or the network breaks down precisely then, between the creation of the TCP connection and the acknowledgment of the HTTP `get` request, the `write` system call issued by the HTTP client times out when the manager's retransmission timer goes off (we already investigated how this timer works in Section 7.5.1.1 and Section 7.5.1.3). The TCP connection is then automatically broken by the kernel. In this case, the HTTP client does nothing else than report a failure to the collector. The reconnection strategy that we recommend to follow is the same as that recommended when an already established TCP connection is timed out by the manager: three retries and two types of alarm.

7.6 HTTP-Based Communication for Pulled Data: WIMA-CM-pull

Now that we have covered in length the complex case of push-based regular management, let us present the much simpler case of pull-based *ad hoc* management.

In the case of pull, the manager creates a new TCP connection and issues one HTTP `get` per retrieved management data (e.g., per SNMP MIB variable or per CIM object). Once the data is retrieved, the TCP connection is torn down by the manager. Everything is initiated from the manager side (creation of the connection and client-server request), so we have no problems with firewalls (no reversed client and server roles). The connection is not persistent, so we need not perform timeouts and reconnections. If a transmission fails, no retry is performed automatically. We let the interactive user (administrator or operator) decide whether a new attempt should be made.

Several data retrieved in a row require the creation of several connections. In the case of SNMP, we cannot retrieve a varbind list (vector of MIB variables) with a single HTTP `get`, but only one MIB variable. The priority here is to keep the communication model simple, not to make it efficient, because by definition *ad hoc* management occurs rarely (it is typically used for troubleshooting). This design is similar to the way the Web operates today with HTTP/1.0 servers.

The MIME headers that we use in WIMA-CM-pull are the same as those defined in WIMA-CM-push, as depicted in Fig. 34. We use the same content type `application/mgmt`, with the same parameters: *mapping* and *version*. As a result, WIMA-CM-pull is also independent of the information model: we can transfer SNMP data, CIM data, etc.

1. This value is indicative. The components making up the management application should allow the number of retries to be configured by the administrator.

The syntax of the URLs accessed by the manager follows a very simple rule. On the agent, we have a program called `get`; it can be implemented as any kind of CLHS. This program takes a single argument in input: the identifier of the data that we want to retrieve. In the case of SNMP, this identifier is called `oid` and corresponds to the OID of the MIB variable. For instance, if the manager wants to retrieve an SNMP MIB variable from MIB-II, it issues an HTTP `get` request for a URL structured as follows:

<`http://agent.domain:280/mgmt/subscribe/snmpv1/mib-2/get?oid=A.B.C.D`>

where `agent.domain` is the fully qualified domain name of the agent (as defined in Chapter 6), and `A.B.C.D` is the OID of the SNMP MIB variable that the manager wants to retrieve. The header of the agent's reply looks like this:

```
HTTP/1.1 200 OK
Date: Wed, 10 May 2000 02:42:57 GMT
Server: Apache/1.2.4
Last-Modified: Thu, 23 Mar 2000 08:58:33 GMT
Mime-Version: 1.0
Content-Type: "application/mgmt"; mapping="SNMPv2-Trap-to-BER"; version="2"

Data
```

Fig. 34. Pull: HTTP header of the agent's reply

The sole difference with WIMA-CM-push is that we do not use MIME multipart to structure the agent's reply in WIMA-CM-pull.

7.7 Summary

In this chapter, we have presented the second core contribution of this Ph.D. work: the communication model of our Web-based integrated management architecture, WIMA-CM. To begin with, we detailed our main design decisions for push-based regular management: the dissociation between the information and communication models, which allows us to transfer SNMP, CIM, or other management data; the use of persistent TCP connections between the manager and all the agents in its management domain; and the implications of the possible presence of a firewall (reversed client and server roles, and creation of the connections by the manager). Then, we explained why we use HTTP to communicate between the manager and the agent, and how we structure the infinite reply of the agent with MIME multipart. We addressed the issue of urgent notifications and detailed the headers of the different MIME parts. We concluded the push-part of WIMA-CM by studying the tricky and multifaceted problem of timeouts and reconnections, and by making several recommendations. Finally, we presented a much simpler communication model for pull-based *ad hoc* management, where the priority was simplicity rather than efficiency.

Chapter 8

XML IN INTEGRATED MANAGEMENT

In the previous chapter, we did not require that the communication model rely on a specific scheme for representing management data. All schemes are allowed in WIMA-CM, including BER, strings, serialized Java objects, HTML, and XML. In this chapter, we explain why one scheme in particular, XML, is better suited than the others for managing IP networks and systems and, more generally, for supporting distributed integrated management at large—that is, the integration of distributed network, systems, application, service, and policy management. The main advantages of using XML to represent management data in persistent HTTP/TCP connections lie in its flexibility, its simplicity, and its property of being a standard, ubiquitous technology (as are HTTP and MIME). We show that XML is especially convenient for distributing management, for dealing with multiple information models, and for supporting high-level semantics.

This chapter is organized as follows. In Section 8.1, we analyze why XML is useful in NSM. In Section 8.2, we explain how to use XML for representing management data. In Section 8.3, we describe how to deal with multiple information models with XML. In Section 8.4, we give examples of the higher-level semantics allowed by XML in agent-to-manager and manager-to-manager communication. In Section 8.5, we describe how XML unifies management-data transfers for all integrated management areas. Finally, we summarize this chapter in Section 8.6.

8.1 Why Use XML in NSM?

The eXtensible Markup Language (XML) has been all over the press since late 1998. Beyond the hype, what is so special about it? And more specifically, what advantages does it bring to NSM?

8.1.1 Overview of XML

XML has been, and is still, developed under the supervision of the W3C. Its building blocks are currently the XML 1.0 specification [230], issued in February 1998, and the Namespaces specification [231], issued in January 1999. XML was initially devised as a document-interchange format for Web publishing. It primarily corrected several flaws in HTML [87, pp. 14–17] and was meant to gradually replace HTML on the Web. But developers of Web browsers have proved to be slow to add support for XML, and the replacement of HTML

is not expected to happen before long. In Web publishing, XML complements HTML more than it replaces it, e.g. to represent complex biomolecules or mathematical equations (see CML and MathML in Section 8.1.2).

The real success of XML was not encountered in Web publishing, though. It soon turned out that XML was also good for structuring and unambiguously identifying complex data structures that may never be viewed or printed, but simply exchanged between applications running on distant machines [32]. This explains why XML is successful in software engineering at large, and not simply in the niche market of Web publishing. XML is fast becoming the *lingua franca* between distant applications that do not speak the same language, that are not developed by the same vendor, and that do not necessarily run on top of the same middleware, or the same virtual machine, or the same operating system. More and more, the HTTP/XML combination is becoming the standard way to implement a three-tier architecture in the software industry.

Strictly speaking, XML is a metalanguage, that is, a language that describes other languages. By extension, we call *XML document* any document expressed in a markup language defined in XML. XML documents have a logical structure (elements) and a physical structure (entities) [87, 32]. Entities can be named, stored separately, and reused to form elements. A Document Type Definition (DTD) is one way of specifying the elements allowed in a particular type of document. XML documents are normally transferred via HTTP. Information held in XML format is self-describing and allows different viewers to render it differently (as sound, graphics, text, etc.), or different users to customize its rendering by the same viewer. For more information on XML, the reader is referred to Goldfarb and Prescod [87] for introductory material and Megginson [144] for advanced material.

8.1.2 Who uses XML?

Although XML is a fairly recent technology, it is already used in (or pervades) many areas. For instance, all major relational databases already support (or have announced the support for) an XML API. There are also regularly reports in the press that the Electronic Data Interchange (EDI) industry is gradually being taken over by XML, which allows for dramatic cost cuts.

Many XML-based markup languages have already been proposed in many sectors of the industry [240, 28], including the following:

- MathML (Mathematical Markup Language), one of the first markup languages written in XML, already supported by most mathematical software tools
- SMIL (Synchronized Multimedia Integration Language)
- SyncML for data synchronization in mobile communications
- XML-QL (XML Query Language), an SQL-like query language for accessing relational databases
- XML/EDI (XML for Electronic Data Interchange)
- DrawML, VML (Vector Markup Language), SVG (Scalable Vector Graphics), and PGML (Precision Graphics Markup Language) for 2D graphics
- HGML (HyperGraphics Markup Language) for wireless access to graphical data on the Web
- SDML (Signed Document Markup Language)
- DSML (Directory Services Markup Language)
- cXML (commerce XML) for business-to-business electronic commerce
- CML (Chemical Markup Language) and BIOML (Biopolymer Markup Language) in the chemical industry
- VoxML (Voice Markup Language) for voice applications
- MusicML (Music Markup Language), a set of labels for notes, beats, and rests that allows compositions to be stored as text but displayed as sheet music by Web browsers
- AML (Astronomical Markup Language) and AIML (Astronomical Instrument Markup Language) in astronomy
- OFX (Open Financial Exchange) and tpaML (Trading Partner Agreement Markup Language) in the finance industry

8.1.3 Advantages of using XML in general

How could XML become so pervasive in so many different areas and in such a short time? The main advantages of using it in the software industry at large (that is, not specifically in NSM) are fivefold. First, XML is easy to learn. Familiarity with CORBA, J2EE, or DCOM platforms and technologies typically requires half a year of training and practice, whereas XML requires a few weeks. The reason for this is that XML is considerably less elaborate and sophisticated, hence easier to understand and become familiar with. Second, XML is very inexpensive because many XML parsers and editors are already available for free [233]. Middleware platforms, conversely, are generally fairly expensive, especially those supporting CORBA and J2EE. The few existing CORBA platforms that are both feature rich and free are seldom used in industry for maintenance reasons. Third, XML allows software developers to use the same technology everywhere, whatever the application domain. This results in significant cost savings for software-development project managers. In this respect, XML is as successful as HTTP, Web browsers, and Java applets before it. Fourth, XML documents are human readable, and not simply machine readable (unlike BER-encoded documents, for instance). This simplifies debugging significantly. Fifth, because it is so easy to learn, simple to use, and inexpensive, the industry has adopted it as *the* way to interface with legacy systems. This feature is particularly useful when two companies A and B merge: there has to be an old application somewhere (e.g., a proprietary database) in company A that cannot be ported to a modern system at a reasonable cost, or within a reasonable time, but still has to communicate with its counterpart in company B. XML is well suited for that.

8.1.4 Advantages of using XML in NSM

As XML is becoming ubiquitous, the number of people who are familiar with it is growing fast. It therefore makes sense to capitalize on this know-how in NSM as well. But beyond this sole reusability of technical expertise, and beyond the general-purpose advantages of XML listed above, XML also exhibits numerous advantages in the specific case of NSM. We identified five major advantages that could soon make XML become the *lingua franca* in NSM.

A truce in the middleware war

The first and foremost strength of XML in NSM is that it brings a truce to the middleware war. Object-oriented DPEs such as CORBA, Java, or DCOM cause a number of problems that we summarized in a catchphrase in Section 4.4: the *my-middleware-is-better-than-yours* syndrome. Compared to these DPEs, XML keeps a low profile. It does not offer a full-blown object-oriented DPE. Instead, when combined with HTTP, it offers a lightweight, inexpensive way of exchanging structured data between distant applications. Both XML and HTTP/1.0 are lightweight, ubiquitous, and inexpensive[1], and combining the two offers a very appealing solution to transfer management data in the IP world.

In short, the HTTP/XML combination is an efficient cure against the *my-middleware-is-better-than-yours* syndrome.

Right kind of semantics for NSM

Before the days of Web-based management, NSM was struggling between two extremes: the poor semantic richness of SNMP MIBs, which is partly due to the poor expressiveness of SMI, and the throes of the *my-middleware-is-better-than-yours* syndrome. HTTP/XML-based distribution allows for the right kind of semantics in NSM: low enough compared to full-blown, object-oriented DPEs, and high enough compared to mere strings or BER.

When the edges (agent and manager) are object oriented, HTTP/XML, viewed as a middleware for distribution, is more general-purpose than Java, CORBA, or DCOM. In particular, we will see in Section 8.4.4 that XML

1. HTTP/1.1 is also ubiquitous and inexpensive, but it does not qualify for “lightweight”.

makes it possible to mimic object-oriented distribution by transferring state as XML data and by allowing the manager to invoke methods on remote objects running on the agent, provided that the manager knows the object naming scheme used by the agent.

Low footprint on agents

The third advantage of using XML in NSM is that it has a reasonably low footprint on agents. This is of paramount importance in the IP world, as we saw in Section 4.4. CORBA has a large footprint, so has DCOM, which makes them inadequate for managing IP networks and systems. As for Java, J2ME (Java 2 Micro Edition) has a small footprint, J2SE (Java 2 Standard Edition) has a medium footprint, and J2EE (Java 2 Enterprise Edition) has a large footprint on agents. This advantage in NSM is a direct consequence of HTTP/XML being a low-profile middleware.

Mobile code

Fourth, XML supports simple forms of mobile code such as remote evaluation and code on demand (defined in Section 3.1.6.1). For instance, a script can be transferred from a manager to an agent (or from a top-level manager to a mid-level manager in distributed hierarchical management) between `<script>` and `</script>` tags, and attributes can convey metadata such as the name and version of the scripting language. This enables administrators to smoothly integrate weakly and strongly distributed hierarchical management paradigms over time, as agents get smarter and can bear an increasing proportion of the management-application processing. This advantage is very important for the next management cycle: it leaves a door open for the future, once the security issues inherent to mobile code are better understood and addressed. The XML support for simple forms of mobile code is also part of the “right kind of semantics” mentioned earlier.

Dealing with the heterogeneity of information models

Fifth, XML copes with the heterogeneity of the information models: SNMP, CIM, OSI, etc. We explained this in Chapter 7, and will devote Section 8.3 entirely to this issue.

8.1.5 Advantages of using XML in integrated management

XML has not only the potential to become the *lingua franca* in NSM, but also in integrated management at large. XML can be used for everything, not only network and systems management, but also application, service, and policy management. The HTTP/XML combination is a true enabler of the so-called “global enterprise management” vision that marketing people have been trumpeting for years. XML is not a representation scheme dedicated to a specific management function: it is a general-purpose means of representing self-describing data. In the IP world, we have never been so close to integrating management in the past. We will come back to this in Section 8.5.

Note that WIMA is not the sole management architecture leveraging HTTP/XML: so does WBEM, since the proprietary HMMP protocol initially envisioned was dropped in favor of the standard HTTP and XML technologies.

8.2 XML for Representing Management Data: Model- and Metamodel-Level Mappings

Once an NSM information modeler has decided to use XML to represent management data, the very first problem he/she is faced with is the way SNMP MIB variables, CIM objects, etc. should be mapped to XML. There are basically two ways of representing management data in XML: at the model and metamodel levels. In DMTF parlance, these are respectively called the schema and metaschema mappings. So far, this important issue has been only briefly touched upon by the DMTF in a slightly cryptic style [67 p. 6, 64 pp. 9–10] and has been eluded by the IETF. In this section, we strive to clarify the strengths and weaknesses of these two different approaches, which are both allowed in WIMA and present different advantages.

8.2.1 Model-level mapping

A model-level mapping is one in which the DTD is specific to a particular SNMP MIB (set of MIB variables), CIM schema (set of CIM classes), or what we generically call a virtual management-data repository. The XML elements and attributes in the DTD bear the same names as the SNMP MIB variables, CIM classes and properties, etc. A simple example of model-level mapping is the following:

```
<interface>
<bandwidth type=”string”>100 Mbit/s</bandwidth>
</interface>
```

8.2.1.1 Strengths and weaknesses of model-level mapping

The main advantage of a model-level mapping is that the DTDs and the XML documents that comply with these DTDs are easy for a human to read. They resemble the examples typically found in beginner’s guides to XML, with a nice and intuitive containment hierarchy. They are also simple to parse and render graphically (e.g., with a Web browser). Another advantage is what the DMTF calls the *validation power*: validating XML parsers can perform in-depth checks (e.g., type checking) on XML documents.

The main disadvantage of a model-level mapping is that we need many DTDs, one per SNMP MIB or CIM schema. This can be a problem for agents that are resource constrained but support many SNMP MIBs, CIM schemata, etc. This is less of a problem on the manager side, as we can easily increase the amount of memory if need be. Another weakness is that the translation from SMI definitions for an SNMP MIB—or Managed Object Format (MOF) definitions for CIM schemata—to XML is not easy to automate, because the logics of hierarchical containment of XML elements does not always directly map onto the logics of containment of SMI, MOF, etc. A third problem, specific to CIM (which is object-oriented), is the possible explosion of the namespace on the agent side if we define too many classes. Dealing with a large namespace on the manager side is less of a problem, as stated before.

8.2.1.2 Example: SNMP-MIB-to-XML mapping

Let us illustrate the concept of model-level mapping by considering a simple example in the SNMP realm: the mapping of MIB-II to XML. MIB-II is *the* standard SNMP MIB, and is supported by almost all SNMP-aware network devices and systems. For the sake of conciseness, let us consider only the Interfaces Group in MIB-II (given in Appendix C). Fig. 35a shows a DTD for this Interfaces Group[1]. Fig. 35b gives an example of XML document that complies with this DTD. To keep these figures reasonably small, we did not represent metadata related to access control, description, etc.

1. We do not claim that this DTD is the best for this example. As usual with XML, there is often a thin line between an element and an attribute, so several DTDs would make sense for the MIB-II Interfaces Group.

```
<!-- Parameter entities -->
<!ENTITY % statusValues “(up | down | testing)”>
<!ENTITY % octetsTable “(unicastPackets?, nonUnicastPackets?, discards?, errors?)”>
<!-- We only consider the Interfaces Group here -->
<!ELEMENT mib-2 (interfacesGroup?)>
<!ELEMENT interfacesGroup (numberOfInterfaces?, interfacesTable?>
<!ELEMENT numberOfInterfaces (#PCDATA)>
<!ELEMENT interfacesTable (interface+)>
<!ELEMENT interface (description?, type?, mtu?, speed?, physicalAddress, status?, lastChange?, inOctets?,
outOctets?, specific?)>
<!ATTLIST interface index (#PCDATA) #REQUIRED>
<!ELEMENT description (#PCDATA)>
<!-- Valid types are specified by RFC1213. To keep this example simple, accept any string of characters-->
<!ELEMENT type (#PCDATA)>
<!ELEMENT mtu (#PCDATA)>
<!ELEMENT speed (#PCDATA)>
<!ELEMENT physicalAddress (#PCDATA)>
<!ELEMENT status EMPTY>
<!ATTLIST status administrative %statusValues; operational %statusValues;>
<!ELEMENT lastChange (#PCDATA)>
<!-- Use a parameter entity for inOctets and outOctets -->
<!ELEMENT - - inOctets (%octetsTable;?, unknownProtocols?)>
<!ELEMENT unicastPackets (#PCDATA)>
<!ELEMENT nonUnicastPackets (#PCDATA)>
<!ELEMENT discards (#PCDATA)>
<!ELEMENT errors (#PCDATA)>
<!ELEMENT unknownProtocols (#PCDATA)>
<!ELEMENT - - outOctets (%octetsTable;?, queueLength?)>
<!ELEMENT queueLength (#PCDATA)>
<!ELEMENT specific (#PCDATA)>
```

(a) DTD

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE SNMP-MIB-II-INTERFACES-GROUP SYSTEM “snmp-rfc1213-excerpt.dtd”>
<push cycle=”4567” localTime=”Fri, 7 Jul 2000 10:55:05 +0200” frequency=”600”>
<mib-2 oid="1.3.6.1.2.1">
<interfacesGroup>
<interfacesTable>
<interface index=”1”>
<status administrative=”up” operational=”up”/>
<inOctets>
<unicastPackets>
80000
</unicastPackets>
<errors>
12
</errors>
</inOctets>
</interface>
</interfacesTable>
</interfacesGroup>
</mib-2>
</push>
```

(b) Example of XML document

Fig. 35. Model-level XML mapping of the Interfaces Group in SNMP MIB-II

The definition of the MIB-II Interfaces Group is written in SMIv1 and available in Appendix C. If we translate these SMI definitions into XML, several transformations are natural because we do not want XML documents to be tweaked by SMI idiosyncrasies, such as the impossibility of having nested tables. For instance, four MIB variables are identical for inbound and outbound octets: the number of unicast packets, the number of non-unicast packets, the number of discards, and the number of errors. Rather than duplicating them as in

MIB-II, it makes sense to group them in the DTD into a single parameter entity (`octetsTable`). Another example is the similarity between the *administrative status* and the *operational status*. These two MIB variables are closely related, have identical semantics, and can take only three valid values. It therefore makes sense to create a single element (`status`) with two optional attributes (`administrative` and `operational`) and a single set of attributes stored in a parameter entity (`statusValues`). By making such transformations, we get a DTD that is easy to read for a human (see Fig. 35a), and XML documents complying with this DTD are also easy to read (see Fig. 35b).

Note that all the XML elements in Fig. 35a are optional (this is epitomized by the question marks in the DTD), except `interface` and `index`, which are obviously mandatory. This allows a manager or an agent to transfer any number of MIB variables in a single XML document.

In Fig. 35b, we consider the example of data pushed by an agent. This XML document complies with the DTD specified in Fig. 35a. In this example, the administrator subscribed the manager for the administrative and operational statuses of the agent's interface #1, as well as the number of inbound unicast packets and the number of inbound errors (for more details about the semantics, see Appendix C).

8.2.1.3 Example: CIM-schema-to-XML mapping

In the WBEM/CIM realm, let us consider a simple schema consisting of a single CIM class. This class is defined in Appendix D and comes from a DMTF white paper [64]. We will use the same class in Section 8.2.2.3 when we consider metamodel-level mappings. An example of model-level mapping of this CIM class is given in Fig. 36:

```
<!ELEMENT class (caption, description, installDate, name)>
<!ATTLIST class className (#PCDATA) #REQUIRED status (OK | Error | Degraded | Unknown)>
<!ELEMENT caption (#PCDATA)>
<!ATTLIST caption type="string" #FIXED maxlen="64" #FIXED>
<!ELEMENT description (#PCDATA)>
<!ATTLIST description type="string" #FIXED>
<!ELEMENT installDate (#PCDATA)>
<!ATTLIST installDate type="datetime" #FIXED>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name type="string" #FIXED>
```

(a) DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SIMPLE-CIM-CLASS SYSTEM "simple-cim-class.dtd">
<cimclass classname="CIM_ManagedSystemElement" status="OK">
<caption maxlen="64">
This is my caption
</caption>
<description>
This is my description
</description>
<installDate>
"Fri, 7 Jul 2000 10:55:05 +0200"
</installDate>
<name>
This is my name
</name>
</cimclass>
```

(b) Example of XML document

Fig. 36. Model-level XML mapping of a simple CIM class

We see that the class name and property names explicitly appear in the DTD and XML document. We also notice that the XML document is fairly concise and its structure is very simple to understand.

8.2.2 Metamodel-level mapping

A metamodel-level mapping is one in which the DTD is generic and identical for all SNMP MIBs, all CIM schemata, or more generically all the virtual management-data repositories specified with the same metamodel. In other words, there is only one DTD per metamodel (SMIv1, SMIv2, CIM metaschema, etc.). The XML elements and attributes in the DTD bear generic names such as *class*, *property*, *operation*, and more generally all the keywords defined by the metamodel. With a metamodel-level mapping, the simple example presented at the beginning of Section 8.2.1 becomes:

```
<class name=”interface”>
<property name=”bandwidth” type=”string”>
<value>100 Mbit/s</value>
</property>
</class>
```

8.2.2.1 Strengths and weaknesses of a metamodel-level mapping

The main advantage of a metamodel-level mapping is that the translation from SMI definitions (for an SNMP MIB), MOF definitions (for CIM schemata), and so on to XML is straightforward to automate. This makes it easy to write external or internal management gateways for legacy systems. This is an important factor during the migration phase from SNMP-based to CIM-based management (or from SNMP- to WIMA-based management, see Section 6.3.6 and Section 6.4.4). Another strength is that by having a single DTD for all SNMP MIBs, CIM schemata, etc., we reduce the overhead for the agent as well as the manager. In particular, for CIM, the memory footprint of the namespace is guaranteed to remain reasonably small because the number of metamodels supported by an agent is always limited.

The main disadvantage of a metamodel-level mapping is that the DTDs are difficult (sometimes *very* difficult) for humans to read and can only reasonably be interpreted by machines. This makes debugging cumbersome. Graphical rendering is also more complex and less user-friendly. Another important problem is validation: with metamodel-level mappings, XML parsers can do very basic validation because they operate at a level of abstraction where they see classes, properties, method names, and all sorts of very generic elements. This makes it easy for a manager or an agent to send a valid but nonsensical XML document. Another weakness of metamodel-level mapping is that XML documents are significantly more verbose than with a model-level mapping; we will illustrate this in the next two examples. A metamodel-level mapping increases network overhead (XML documents are significantly longer) and slightly increases latency (parsing time, marshalling time, unmarshalling time).

8.2.2.2 Example: SMI-to-XML mapping

In the case of SNMP, two examples of SMI-to-XML mappings can be found in the literature. The first was published by John *et al.* in October 1999 [114]. The DTD that they propose for SMIv2 is very basic and consists only of `ELEMENT` definitions (no entities, no attributes, not even for constant strings). To be fair to the authors, this DTD was not the central part of their work, which focused on the specification of XNAMI, an XML-based management architecture that supports dynamically modifiable MIBs.

The second example is a comprehensive SMI-to-XML mapping issued in June 2000 by Schönwälder and Strauss [186]. In their Internet-Draft, they propose a way to use XML to exchange SMIv1, SMIv2, and SMIng module definitions (that is, descriptions of SNMP MIB variables) between XML-enabled applications. The DTD specified in their Internet-Draft has been integrated into the SimpleWeb IETF MIB converter [193] maintained by the University of Twente, The Netherlands and the Technical University of Braunschweig, Germany. The resulting representation of the MIB-II Interfaces Group in XML is available in Appendix D. The Interfaces Group alone takes 10 pages, which gives a clear indication of the verbosity of this type of mapping. We can also see that a lot of information is duplicated, just as in MIB-II, e.g. the values that the administrative

and operational statuses can take, or the managed objects that are similar for inbound and outbound traffic. This XML document was generated automatically and almost instantaneously by the SimpleWeb converter, which demonstrates how easy it is for a machine to translate SMI into XML.

8.2.2.3 Example: CIM-metaschema-to-XML mapping

In the case of CIM, many examples of metaschema mappings are available from the DMTF. In Appendix D, we reproduce an example of CIM-to-XML mapping done at the metamodel level. This example comes from a DMTF white paper. We notice immediately that the readability of this XML document is very poor for a human. It takes quite some time to get accustomed to the extra level of indirection caused by the metamodel-level mapping, which makes debugging rather difficult. We also see how limited validation is with this type of mapping. Finally, it is obvious that this XML document can be generated automatically by a CIM-aware agent.

8.2.3 Comparison between model- and metamodel-level mappings

If we compare Appendix D with Fig. 35 (MIB-II Interfaces Group) and Appendix D with Fig. 36 (simple CIM class), we see immediately that XML documents complying with a metamodel-level mapping are less readable and more verbose than XML documents complying with a model-level mapping. This is not simply due to the fact that the two mappings in appendix are more detailed and comprehensive, but rather to the very nature of these two types of mapping.

If we analyze Schönwälder and Strauss's DTD [186] and compare it with our partial DTD listed in Fig. 35a, we also clearly see that the former is hampered by SMI idiosyncrasies; e.g., columnar objects are mapped to `column` XML elements. Although this makes automated translation very easy, it significantly hampers human understanding and generates a nonintuitive containment hierarchy of XML elements. Our proposed containment hierarchy in Fig. 35 is much more natural and easy to understand.

In short, the model- and metamodel-level mappings both have their advantages and disadvantages. We cannot say that one is always superior to the other, so both are allowed in WIMA. The choice between these two types of mappings is necessarily a trade-off, and depends on the criteria that are of most importance to the management-application designer. These criteria are often site specific. In this section, we identified six:

- readability of the XML document for a human (vs. a machine)
- verbosity vs. conciseness of the XML document
- automated translation to XML by a management gateway
- validation power
- one DTD vs. numerous DTDs
- memory footprint of the namespace (CIM).

8.3 XML for Dealing with Multiple Information Models

In Section 7.4.3, we explained that management data in transfer (that is, when it is not stored as in-memory data structures by the agent or the manager) can be represented in many ways in WIMA: not only XML, but also HTML, plain strings, BER, serialized Java objects, etc. Because management data encoded in XML documents is self-describing, XML is particularly well suited to cope with multiple information models.

In the example depicted in Fig. 37, we have two management domains. In the left domain, the mid-level manager #1 and all the agents support the SNMP information model; in the right domain, the mid-level manager #2 and all the agents support the CIM information model. (Note that the agents are not represented in Fig. 37.) The administrator needs to integrate management data coming from these two domains, e.g. to work out the root cause of problems occurring at the boundary of these domains, or to perform global statistics on

the collected data). Let us denote XXX the information model that the administrator has decided to adopt to integrate management data. By definition, XXX is site specific.

Translator components are in charge of converting specific information models (SNMP and CIM in our example) into XXX. These components might be integrated into the management server of each mid-level manager, or might run on a separate machine. The communication between the event translators (respectively, the pushed-data translators) and the notification/event filters (respectively, the pushed-data interpreters) can rely on HTTP/XML, especially if the translators run on a separate machine; they can also use another, more efficient interprocess mechanism (e.g., shared memory) for scalability reasons if the components run on the same machine. This alternative is depicted by the string "`??XML??`" on Fig. 37. As for the communication between the event translators (respectively the pushed-data translators) and the XXX event correlator (respectively the XXX data repository), it relies on HTTP/XML.

The issue of translating from one information model to another for the purpose of integration is complex and has been dealt with by other authors, especially Mazumdar [138, 139, 140] and Rivière [174, 175].

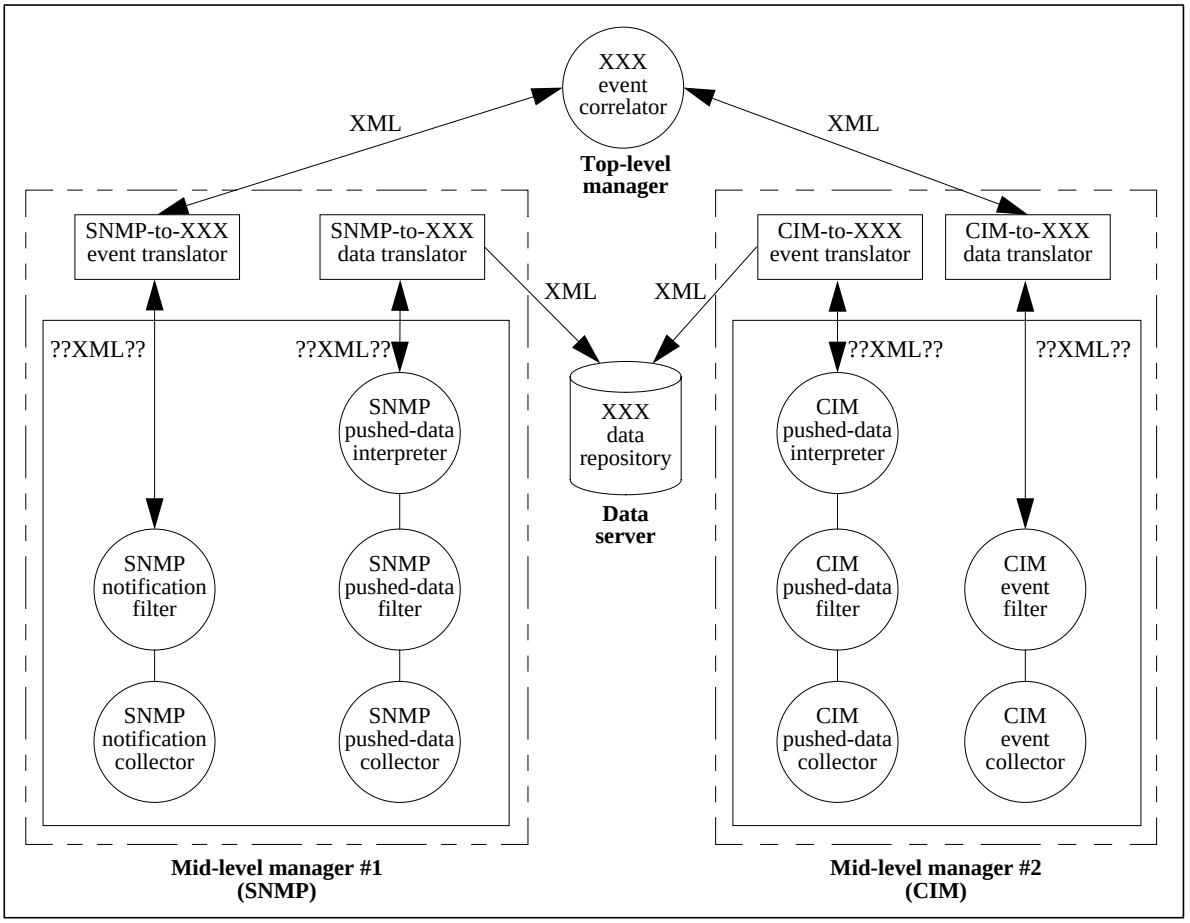

Fig. 37. Multiple information models: generic integration

The approach we just described is called *generic integration*. Whatever XXX, whatever the information model used within a management domain, we always go via translators for both events and data. The main advantage of this solution is that the support for a new information model simply requires that new translators be written, which is elegant in terms of design. The main problem is that XXX is likely to be one of the information models already used by one of the mid-level managers (SNMP or CIM in our example), so we are unnecessarily inefficient for this information model (going through a level of indirection unnecessarily adds some latency).

This problem is solved by resorting to *specific integration* (see example depicted in Fig. 38). This time, we no longer have translators on all mid-level managers, nor an external event correlator. Instead, the administrator

chooses his/her preferred information model (in our example, CIM) and runs translators only on the mid-level managers that support another information model (in our example, the SNMP management server). The main advantage of this solution is that it is globally more efficient than the previous. The chief disadvantage is that it is less generic. If the administrator decides to change the preferred information model (e.g., after the merger of two companies), specific integration requires significantly more work than generic integration. As usual in the software industry, there is a trade-off to be found between modularity and efficiency. As this decision is inherently site specific, one could expect that component-software developers implement both solutions and let the administrator decide whether efficiency should prevail over design modularity, or *vice versa*.

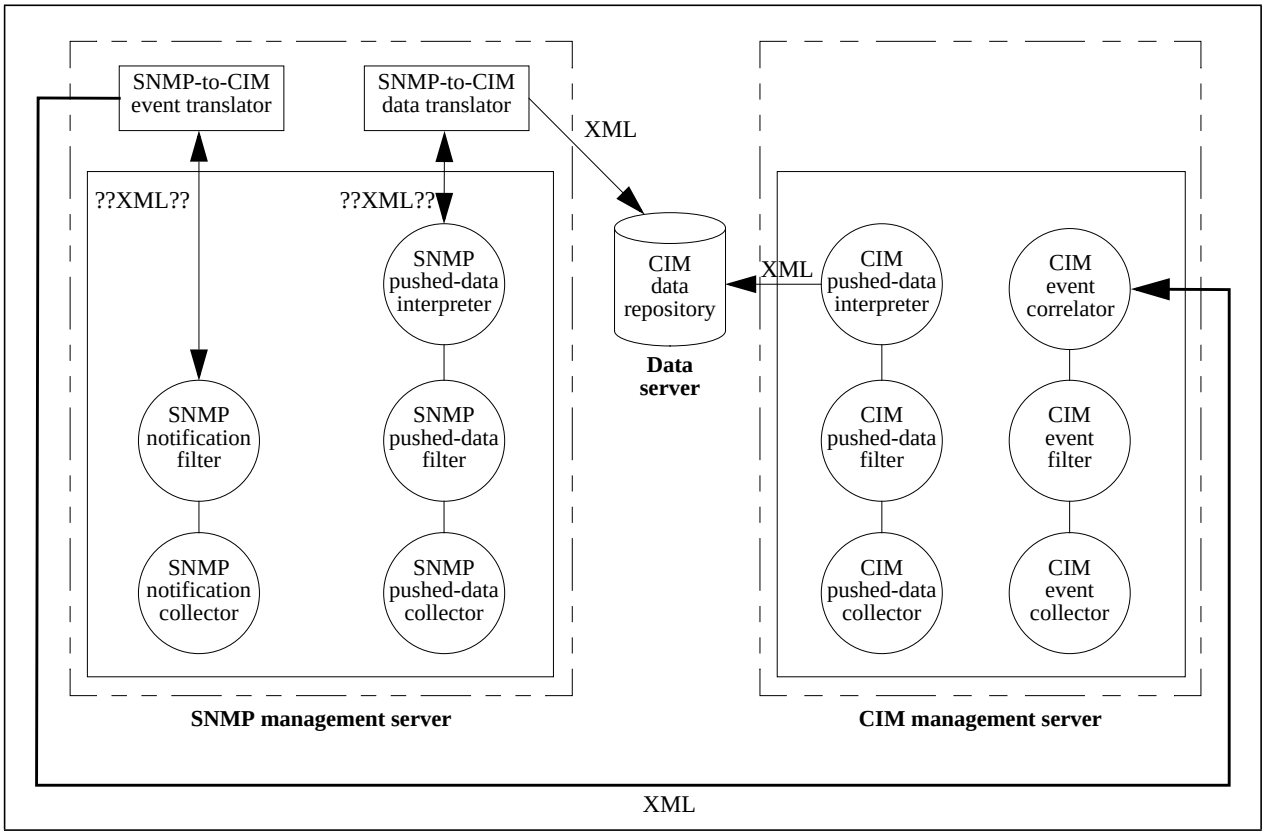

Fig. 38. Multiple information models: specific integration

8.4 XML for High-Level Semantics

In NSM, the level of semantics and the expressiveness of management data are closely related. The level of semantics depends on the abstractions modeled in the virtual management-data repository (SNMP MIB, CIM schema, etc.). It is a characteristic of the model. Conversely, the level of expressiveness lies in the metamodel. A highly expressive metamodel gives the information modeler ample latitude to define abstractions that are useful to the administrator. A poorly expressive metamodel would, for instance, only allow for integers and strings, thereby seriously constraining the definition of appropriate abstractions. A high-level semantics in the model demands a high expressiveness of the metamodel. Of course, a highly expressive metamodel does not necessarily lead to a semantically rich model: this depends on the information modeler's skills.

By extension, the semantic richness often designates both of these characteristics. To simplify things, we will therefore say that high-level semantics allow an information modeler to "think" and manipulate abstractions as a human typically does, whereas low-level semantics force him/her to go down to an abstraction level that is more cumbersome to a human: the nuts and bolts of networking and systems, so to say.

WIMA/XML supports a much richer semantics than that offered by SNMP and SMI. It renders easy many tasks that are not so easy in SNMP-based management. We will give three examples in this section: the transfer of an entire MIB table in one bulk, the suppression of “holes” in sparse tables, and the transfer of an entire time series in one bulk. Beyond SNMP, WIMA/XML is also well suited to object-oriented agents and managers. In our last example, we will describe the remote invocation of a method encapsulated by a remote object.

8.4.1 Transfer of an entire SNMP MIB table

In Section 2.4.1.2 and Section 2.4.1.3, we studied the problems caused by the small maximum size of an SNMP message and the difficulty to transfer MIB tables efficiently via the SNMP protocol (when building a request message, the manager must guess the size of the agent’s response). In WIMA in general, and in XML in particular, we have no limit set on the maximum size of an HTTP message: an XML document can be infinitely large. This allows the agent to push an entire MIB table (or even an entire MIB) in one bulk. In Section 8.2.1.2, we already gave an example of XML document showing how to transfer a part of or the whole of MIB-II.

8.4.2 Suppression of “holes” in sparse SNMP tables

In Section 2.4.1.3, we also described the problems caused by the possible presence of “holes” in sparse SNMP tables. With XML, it is very simple to remove these “holes” altogether, thereby relieving the management server from doing the extra computations required to check for the presence of these “holes” and to get rid of them. To do so, we propose to use a new attribute (`errorCode`) and replace the “hole” with an error code, typically `noSuchInstance` in the case of sparse SNMP tables:

```
<value errorCode=”noSuchInstance”>
</value>
```

The possible values of `errorCode` are specific to a given information model (SNMPv1, v2c, or v3 in the above example). A validating XML parser can check the value assigned to `errorCode` against the information model specified in the header of the XML document.

8.4.3 Time series of a MIB variable

In SMI, there is no provision for time series. As a result, there is no simple means in SNMP to store and retrieve an entire time series for a single managed object, such as the inbound error rate of interface #1 sampled every 10 minutes over a period of 24 hours. This is a problem for offline statistical analysis of SNMP data, which typically works on time series. Today, offline statistical analysis requires administrators to reconstruct time series from the collected data, e.g. in the form of temporal MIBs [11]. Another problem is that some SNMP data can be lost, e.g. due to buffer overflows. This leads to “holes” in the reconstructed time series, and these “holes” are sometimes awkward to deal with. For instance, if we build a time series for per-interface inbound traffic, what do linear or quadratic interpolations lead to in the case of bursty traffic? And does it make sense to interpolate the missing value of an error rate?

This problem is easy to solve in WIMA: time series can be stored at the agent and transferred in one bulk via HTTP/XML. Storing time series at the agent is easy because they usually take up little room when stored as in-memory data structures. For instance, a gauge and a timestamp stored as 32-bit integers each require four bytes of memory. Sampled every 10 minutes over a period of 24 hours, such a MIB variable would generate a time series with the following memory footprint on the agent:

$$(4 + 4) \times (60 / 10) \times 24 = 1152 \text{ bytes}$$

For many agents, such a small overhead is negligible, and dozens (sometimes hundreds) of such time series can easily be stored locally. The advantage of storing a time series at the agent is that the agent can then push an entire time series once a day, in bulk. This is useful because during this operation, the agent can carefully

retransmit data if need be, and can even retry an hour later. This makes data transfers very resilient. Another case when this can be particularly useful is when the agent is disconnected from the manager for extended periods of time, because it is switched off for several hours in a row (e.g., a desktop PC or a mobile phone), because it is in a mountainous area not covered by any antenna (e.g., a mobile phone or a roaming PC), etc. Clearly, there is trade-off to be found between keeping time series in volatile memory, where they take few resources but may be lost in case the agent crashes, and keeping them in persistent storage (e.g., EPROM) where they use up more resources but survive a crash.

When the data is stored at the agent, it can be transferred via XML in a simple way, as depicted in Fig. 39. For the sake of readability, we assume here that the first timestamp occurs at the epoch (time zero) and we only represent the first four values and the last; intermediate values are replaced with the string "`[...]`".

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE SNMP-MIB-II-INTERFACES-GROUP SYSTEM “snmp-rfc1213-excerpt.dtd”>
<push cycle=”4567” localTime=”Fri, 7 Jul 2000 10:55:05 +0200” frequency=”86400”>
<mib-2 oid="1.3.6.1.2.1">
<interfacesGroup>
<interfacesTable>
<interface index=”1”>
<inOctets>
<unicastPackets>
<value “timestamp”0”>81000</value>
<value “timestamp”600”>65000</value>
<value “timestamp”1200”>73000</value>
<value “timestamp”1800”>70000</value>

[...]

<value “timestamp”85800”>100000</value>
</unicastPackets>
</inOctets>
</interface>
</interfacesTable>
</interfacesGroup>
</mib-2>
</push>
```

Fig. 39. Time series with XML

Note that the usefulness of time series is not limited to agent-manager interactions. Time series can also be beneficial in distributed hierarchical management, when mid-level managers are required to push daily statistics to the top-level manager. For instance, the top-level manager might want to be informed by each mid-level manager of the number of serious events handled by its event correlator every 15 minutes, over a period of 24 hours. QoS management is another typical example: it is interesting for the top-level manager to get time series of the number of bandwidth reservations that could not be honored by the boundary routers of each management domain.

8.4.4 Distributed object-oriented programming with XML

Outside the SNMP realm, e.g. for CIM, XML elements and entities can be used to implement what we could call the poor man's object serialization and remote method invocation.

The serialization of objects running at the edges of the persistent HTTP/TCP connection (that is, the agent or the manager) must deal with both state and behavior. The serialization of an object state is simple: the state (or *properties* in DMTF parlance) is translated into XML elements and entities, as already described. The serialization of an object behavior (or *methods* in DMTF parlance) can only be achieved in simple cases, e.g. when the methods are implemented with scripting languages (see the simple support for mobile code described in

Section 8.1.4, with the proposed tags `<script>` and `</script>`). XML can also transfer opaque binary data “as is”, which enables the transfer of serialized Java objects, proprietary C++ objects, etc. Note that in the latter case, the internal structure of the transferred object is completely invisible. This prevents XML parsers from validating the data and makes it difficult for a human to read (and debug) the management data in transit.

As for the remote invocation of object methods in XML, an example can be found in the literature: the DMTF’s specification for CIM operations over HTTP [68]. The DMTF’s model is reasonably simple and can be generalized to almost any object-oriented information model. The CIM DTD [66] defines two elements: `<METHODCALL>` for invoking a method on a remote object, and `<METHODRESPONSE>` for receiving the response of this method call. These two elements are hierarchically contained in an operation element. A simple operation can contain a single method call or response. Two elements are defined by the CIM DTD for simple operations: `<SIMPLEREQ>` for the request and `<SIMPLERSP>` for the response. A multiple operation, on the other hand, allows multiple method invocations to be batched together, thereby reducing the number of messages exchanged between the manager and the agent. In the CIM DTD, the corresponding elements are `<MULTIREQ>` and `<MULTIRSP>`. An example of method invocation including a complete request and a complete response is given in Appendix D. The above-mentioned specification also supports some kind of introspection (via *intrinsic methods*) for discovering the subclasses of a CIM class (`EnumerateClasses`), the instances of a CIM class (`EnumerateInstances`), the CIM objects that are associated to a specific CIM object (`Associators`), etc. Extra metadata is available via *qualifiers*.

This example demonstrates that although XML does not directly support remote method invocation in the way CORBA or Java RMI do, it does provide the means of encoding method invocation and parameters, and of checking the type of each parameter at invocation time. Thus, with XML, it is possible to cleanly trigger an action on a distant agent or manager: there is no need to resort to SNMP’s ugly programming by side effect (see Section 2.4.2.2). XML nicely interfaces with object-oriented information models, which offer high-level semantics to management-applications designers—higher than data-oriented information models such as SNMP’s. By using HTTP and XML, we do not incur the problems of object-oriented DPEs, and we have a simple yet sufficient way of interacting with objects at the edges. In short, we get the best of both the object-oriented and non-object-oriented worlds.

8.5 XML for Integrated Management: a Unified Communication Model

Although WIMA allows for many data representation schemes, unifying the communication model around a single scheme brings several advantages: all management data is in the same format: configuration files, performance statistics, policies, etc.; we no longer have to translate between different representation schemes; we can directly store XML data in the data server for offline processing or archival; etc. For all these reasons, other representation schemes should preferably be reserved for interfacing with legacy systems (e.g., it makes a vendor’s life easier to encode SNMP data in BER during the transition phase) or for debugging (e.g., clear-text strings), but new WIMA-compliant management servers and agents should preferably support XML natively[1].

In Section 8.1.5, we claimed that we have never been so close to integrating management in the IP world, now that we have the powerful combination of HTTP and XML. Let us illustrate this with an example, and demonstrate the advantages of having a unified communication model for network, systems, application, service, and policy management.

In the hierarchical tree depicted in Fig. 40, we use the same communication model between all devices and systems: HTTP, MIME multipart, and MIME parts consisting of XML documents. In this example, we have two management domains—a situation typically encountered by geographically dispersed enterprises. Each

1. Note that WIMA does not make it mandatory to use XML, though.

management domain is supervised by a mid-level 1 manager. For the management domain on the left, the mid-level manager is directly in charge of the agents. With agent 1 (e.g., a large file server with multiple interfaces), the mid-level 1 manager performs both network and systems management tasks. With agent 2 (e.g., a mere hub), it only performs network management. For the management domain on the right side, the mid-level 1 manager delegates network management to one mid-level 2 manager and systems management to another mid-level 2 manager. Unlike its counterpart on the left side, the right mid-level 1 manager does not directly interact with the agents in its domain: only the mid-level 2 managers do. For all these interactions, we use the same communication model based on HTTP/MIME/XML.

Some agents can be under the supervision of multiple mid-level 2 managers, e.g. agent 4 in the example depicted in Fig. 40. This sharing of an agent by multiple managers with different roles is classic and results from what Sloman and Twidle call a *domain overlap* [196]. Still, whether an agent is supervised by a mid-level 1 manager, a mid-level 2 manager, or multiple managers, the communication model remains the same: HTTP/MIME/XML.

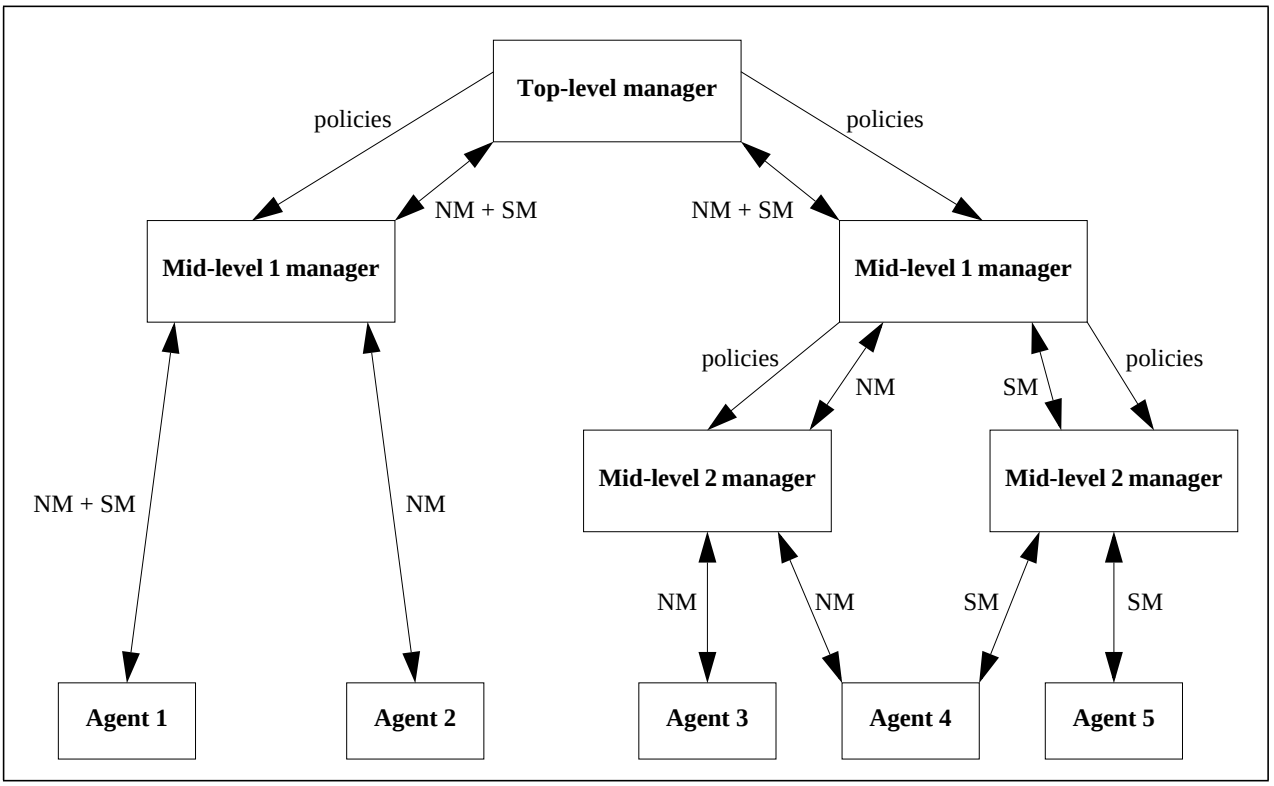

Fig. 40. XML: Hierarchically distributed integrated management

Between all the managers, we have regular exchanges of network and systems management data, all based on HTTP/MIME/XML. Parallel to that, managers exchange policies, that is, high-level abstractions of management translated by managers into lower-level abstractions that can directly be sent to and understood by agents. Policy exchanges also rely on HTTP/MIME/XML. On top of this, we could add application management and service management: they, too, could be based on the same communication model.

In short, this example shows that the combination of HTTP, MIME, and XML offers us a powerful means of unifying the communication model for integrated management.

Note that this unification would not be so straightforward if another representation scheme were used. BER is well suited to SNMP data but very impractical for other information models (see problems described in Chapter 2). Plain strings are good for debugging but are not suitable for elaborate information models such as CIM; neither is HTML, with its fixed tags. Serialized Java objects are not general-purpose enough; neither are binary CORBA objects, vendor-specific C++ objects, etc. In conclusion, the flexibility, simplicity, and small

footprint of XML make it the best suited technology for the integrated management of IP networks and systems.

8.6 Summary

In this chapter, we have shown the advantages of using a communication model based on HTTP/MIME/XML in WIMA. First, we identified the advantages of using XML in general, in NSM, and in integrated management. Second, we explained how to use XML for representing management data and identified the strengths and weaknesses of model- and metamodel-level mappings. Third, we justified why the fact that XML documents are self-describing makes XML particularly appropriate for dealing with multiple information models. Fourth, we illustrated with four examples the high level of semantics offered by XML, which even supports simple forms of distributed object-oriented programming. Fifth, we studied an example demonstrating how XML unifies distributed and integrated management in NSM and beyond. This concludes the third and last core contribution of our Ph.D. thesis.

Chapter 9

A WIMA-BASED RESEARCH PROTOTYPE: JAMAP[1]

In order to validate our new management architecture, we developed in 1998–99 a research prototype called the JAva MAnagement Platform (JAMAP). The main goals of this endeavor were the following:

- to demonstrate the feasibility and simplicity of our push-based organizational model
- to demonstrate the feasibility and simplicity of our communication model
- to highlight the usefulness of advanced Web technologies in NSM
- to show how to deal with agents supporting native SNMP MIBs
- to prove to network-device and system vendors that the support for WIMA is simple

In this chapter, we show that all of these objectives were met. Note that the goal of developing JAMAP was *not* to offer a full-blown management platform for use in a production environment, nor to carry out in-depth performance evaluation, nor to benchmark JAMAP against other management platforms.

This chapter is organized as follows. In Section 9.1, we describe how JAMAP implements WIMA. In Sections 9.2 through to 9.4, we present the detailed design of the three tiers of our management architecture: the management station, the management server, and the agent. In Section 9.5, we show how we made reusability a reality in JAMAP. Finally, we summarize this chapter in Section 9.6.

9.1 Overview of JAMAP

JAMAP is an example of WIMA-based management platform. It implements many of the concepts presented in Chapters 6 and 7. However, it does not support XML because its development predates the work presented in Chapter 8. The motivation behind the development of this research prototype was to demonstrate first of all the feasibility of WIMA, and second its simplicity.

JAMAP should not be considered *the* way to implement WIMA. To begin with, we made a number of simplifications for the sole purpose of reducing the development time and having the prototype ready for an internal demonstration in March 1999. Moreover, some hot software-engineering issues (e.g., “What exactly is a software component?”, or “Is Java appropriate for implementing component software?”) are still very much

1. Part of the material presented in this chapter was published in the proceedings of DSOM’99 [135].

open to debate—and outside the scope of this thesis. JAMAP is just one way of implementing the WIMA management architecture. It does not seek optimality.

Before we begin with the description of JAMAP, we gratefully acknowledge that much of its implementation was performed by Laurent Bovet in the course of his M.S. thesis [30], under the supervision of the author. In case the reader is interested in the internals of JAMAP, the code of release 0.3 is freely available on the Web [136] under the GNU General Public Licence [86].

9.1.1 Key design decisions at a glance

The main novelty demonstrated by JAMAP is the push-based transfer of regular management data, from the agent to the management server, via a persistent HTTP connection structured with MIME multipart. To the best of our knowledge, this constitutes an innovation to manage IP networks and systems.

In order to reduce the development time of JAMAP, we did not implement WIMA in its entirety. JAMAP supports:

- Java-based three-tier architecture: management station, management server, and agent
- publish-subscribe pattern
- push-based distribution for regular management and notification delivery
- pull-based distribution for *ad hoc* management
- rule-based event generation for monitoring
- basic rule-based event filtering for event correlation
- basic event handling
- persistent HTTP/TCP connections with MIME multipart
- compression of management data
- automated reconnections
- management data encoded in serialized Java objects or plain-text strings
- SNMP MIBs

JAMAP does not support:

- bulk transfers of management data (multiple MIB variables per MIME part)
- going across a firewall (no secure relay)
- keepalives
- management data encoded in XML
- CIM schemata

9.1.2 More on the design of JAMAP

The core of JAMAP (agent's push scheduler, persistent HTTP connection, MIME multipart, dynamic data compression, agent's data dispatcher, and manager's data collector) was coded in only two weeks. Most of the coding effort afterward went into implementing the management server, especially the rule-based system that includes a GUI-based rule editor. Overall, the development of JAMAP took a bit more than four man-months. This demonstrates that vendors could rapidly develop WIMA-compliant agents (IP network devices and IP systems). As expected, the development of the management server requires significantly more work. Still, professional-grade WIMA-compliant managers could be available in about six months, which is reasonable.

As far as the information model is concerned, JAMAP supports only SNMP MIBs because they are currently the only virtual management-data repositories widely deployed in IP networks and systems. On the agent side, we tested MIB-II [143] and the Host Resources MIB [237] on a Linux PC. On the manager side, we tested a Linux PC and a Windows PC. Both ran the same code, which proves the portability of Java when developers refrain from using proprietary extensions. JAMAP does not require that native SNMP MIBs be changed in the agents; it interfaces with these MIBs via Java classes written by AdventNet (see Section 9.5).

The components described in Chapter 6 are implemented as Java classes or servlets on both sides of the persistent communication pipe (management server and agent). Servlets use HTTP to communicate with one another, be they on the same or different machine(s). Consequently, servlet-to-servlet communication within the management server is based on HTTP. We do not use object-oriented frameworks *per se* in JAMAP, but we use JDK 1.1.7, which some people consider a framework.

Access to the data repository is based on NFS, a classic in the Unix world. Java objects are directly stored in serialized format, one file per SNMP MIB variable, for convenience. Notifications are not currently archived in JAMAP.

In order to cut the development time, we opted to encode only one management data per MIME part. A push cycle therefore consists of several MIME parts in JAMAP (in WIMA, an entire push cycle is normally sent in a single MIME part). Clearly, this does not demonstrate the power of bulk transfers in WIMA.

For data subscription, we provide the administrator with an SNMP MIB browser. To implement this, we reused the MIB browser and some Java classes that AdventNet makes freely available on the Web (see Section 9.5).

MIME parts carrying SNMP notifications are not compressed because the compression ratio would be poor for so little data, and the increased latency would not be worth the meager savings in network overhead.

Finally, the servlets running on the management server can be distributed over several physical machines. This distribution will be clarified in Section 9.3.

9.1.3 Advanced Java technologies in JAMAP

We use two advanced Java technologies in JAMAP: servlets and serialization. Let us briefly describe them.

Java servlets

JAMAP relies heavily on HTTP-based communication between Java applets and servlets. Servlets [58] only recently appeared on the Web; they are an improvement over the well-known CGI scripts. Unlike CGI scripts, which are typically written in a scripting language like Perl [238] or Tcl/Tk [160], servlets are Java classes loaded in a JVM via an HTTP server. The HTTP server must be configured to use servlets and associate a URL with each loaded servlet. At start-up time, one servlet object is instantiated for each configured servlet. When a request is performed on a servlet URL, the HTTP server invokes a method of the servlet depending on the HTTP method used by the request. All servlets implement one method per HTTP method. For instance, the `doGet()` method is invoked when an HTTP `GET` request comes in for the corresponding URL.

Modern operating systems generally support multithreading. As a result, most HTTP servers now support concurrent accesses. Several HTTP clients may therefore invoke concurrently the same method of the same servlet. This allows the sharing of the same servlet by multiple persistent connections. We used this feature extensively in JAMAP when we tested it with several agents. Like any URLs, Java servlets can also leverage the general-purpose features of HTTP servers (e.g., access control).

During the development of JAMAP, servlet environments were in constant evolution. During our work, Sun Microsystems's specification of the servlets changed from version 2.0 to version 2.1, but public-domain implementations remained at 2.0. For JAMAP, we first used the Apache HTTP server version 1.3.4 and the Apache servlet engine Jserv 0.8. But we had problems because Jserv 0.8 did not support concurrent accesses to servlets and the response stream was buffered (both problems were later corrected in Jserv 1.0). In the meantime, we switched to another HTTP server, Jigsaw 2.0.1, which offered good support for servlets.

Java serialization

Serialization is a feature of Java that allows an arbitrarily complex object to be translated into a byte stream. In JAMAP, we used it for ensuring the persistence of the state of an object and for transferring objects over the

network. Objects containing references to other objects are processed recursively until all necessary objects are serialized. The keyword `transient` can be added to the declaration of an attribute (e.g., an object reference) to prevent its serialization.

For network transfers, instead of defining a protocol, one can use serializable classes dedicated to communication. Such classes offer a `writeObject()` method on one side, and a `readObject()` method on the other. For persistence, serialization proved very useful in JAMAP to store rules and agents configurations.

9.1.4 Overview of the communication aspects

Fig. 41 and Fig. 42 are synthetic views of the communication between the different Java applets and servlets running on the agent, management server, and management station. Fig. 41 depicts push-based monitoring and data collection, while notification delivery and event handling are represented on Fig. 42. The three-tier architecture described in Section 6.2.2 appears clearly in these two figures.

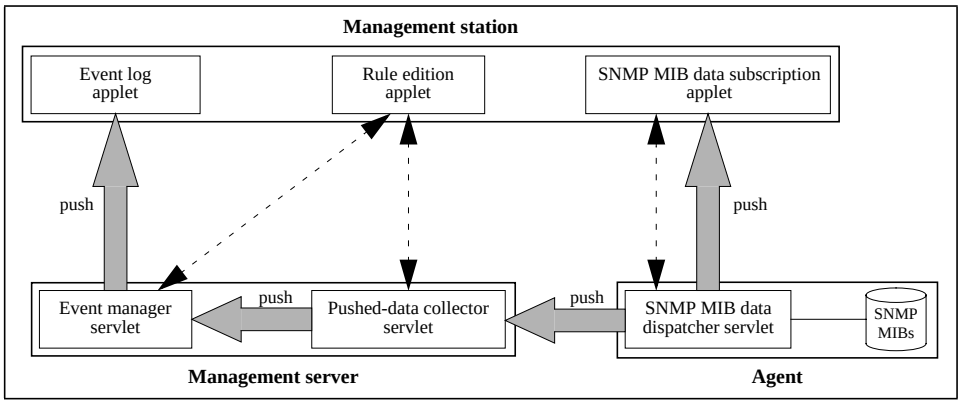

Fig. 41. JAMAP: Communication between Java applets and servlets for monitoring and data collection

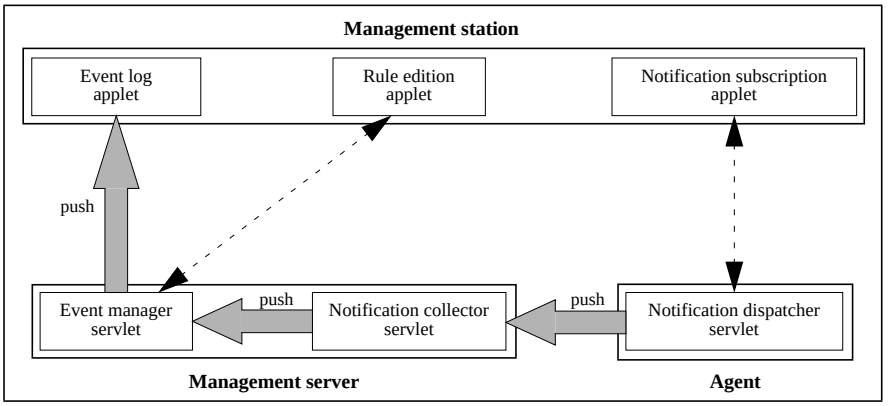

Fig. 42. JAMAP: Communication between Java applets and servlets for notification delivery and event handling

The push arrow between the *SNMP MIB data dispatcher* servlet and the *SNMP MIB data subscription* applet represents the path followed by MIB data retrieved interactively (in attended mode, data can be either pushed or pulled). The other push arrows depict regular management transferred in unattended mode. The dotted arrows represent the applet-to-servlet dialogs that take place at the subscription phase. Note that the same push mechanism is used everywhere: between the agent and the management server, between the management server and the management station, between the agent and the management station, or between the servlets within the management server. Low-level classes also use the same producer-consumer pattern everywhere.

9.1.5 Distribution phase for monitoring and data collection

The distribution phase for monitoring and data collection is depicted in Fig. 43. If we compare this figure with the generic one presented in Chapter 6 for WIMA (Fig. 15, p. 110), the main difference is the presence of Java servlets in JAMAP. They glue together components that are logically related. As mentioned already, servlets communicate via HTTP; so Java objects living in different servlets also communicate via HTTP (e.g., the pushed-data interpreter and the event correlator).

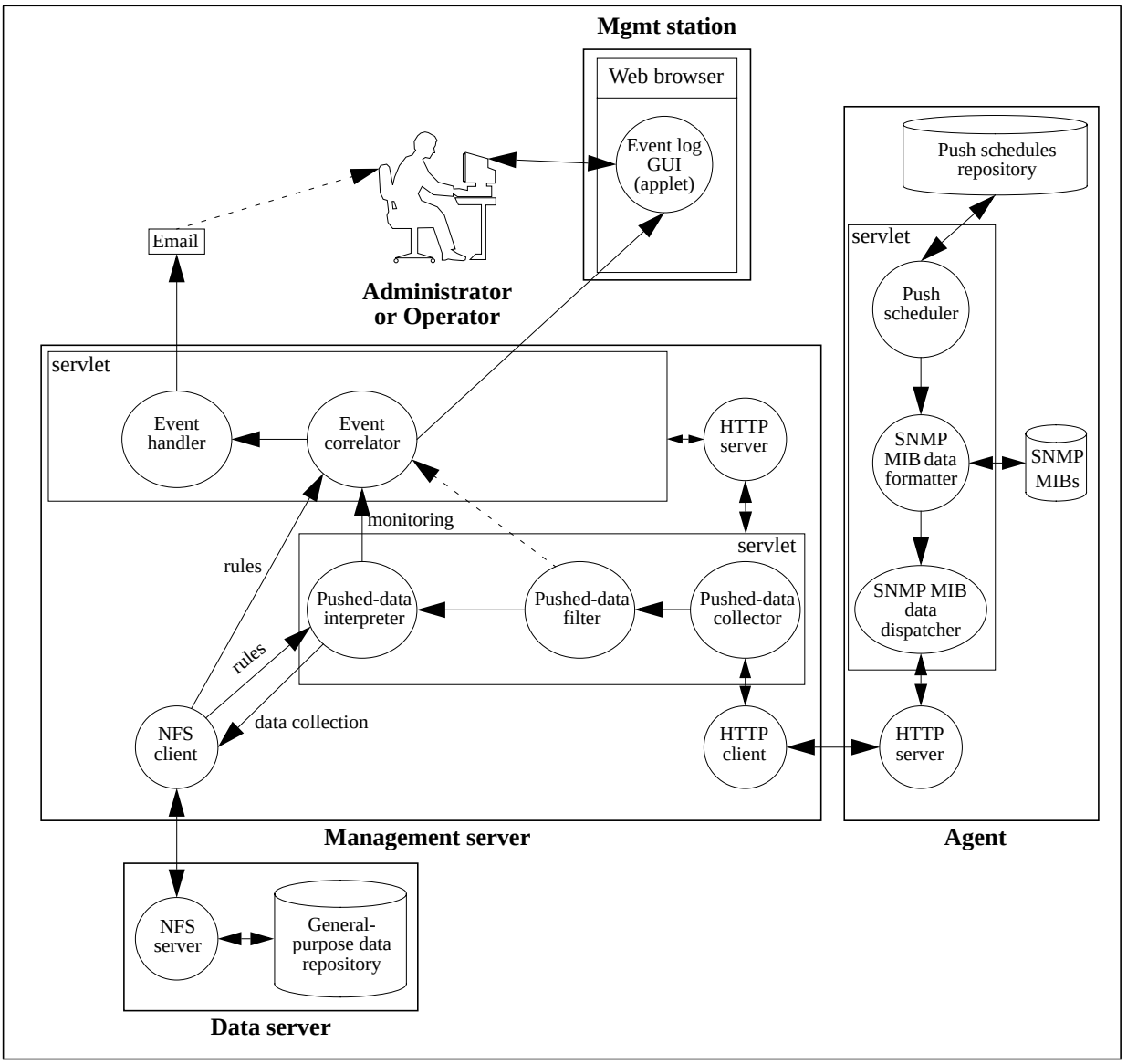

Fig. 43. Push model in JAMAP: monitoring and data collection

Another difference is that we simplified considerably the visualization of events in JAMAP, in order to reduce the development time. Instead of writing a full-blown network-map GUI, aware of the network topology and updating dynamically the colors of the icons representing network devices and systems, we simply implemented a log window called the *event log GUI* applet. In this window, we log one entry per incoming event, line by line. Since we do not use network-map GUIs, we do not have a network-map registry either: the event correlator sends events directly to the log window, which runs permanently in the Web browser of the management station. Consequently, JAMAP currently supports only one management station. (Note that it would be simple to send the same event to multiple log windows running in different Web browsers.)

The third noticeable difference between Fig. 43 and Fig. 15 is that event handlers can only use email to contact the administrator or an operator, and cannot archive events. This was deemed acceptable for a prototype.

9.1.6 Distribution phase for notification delivery

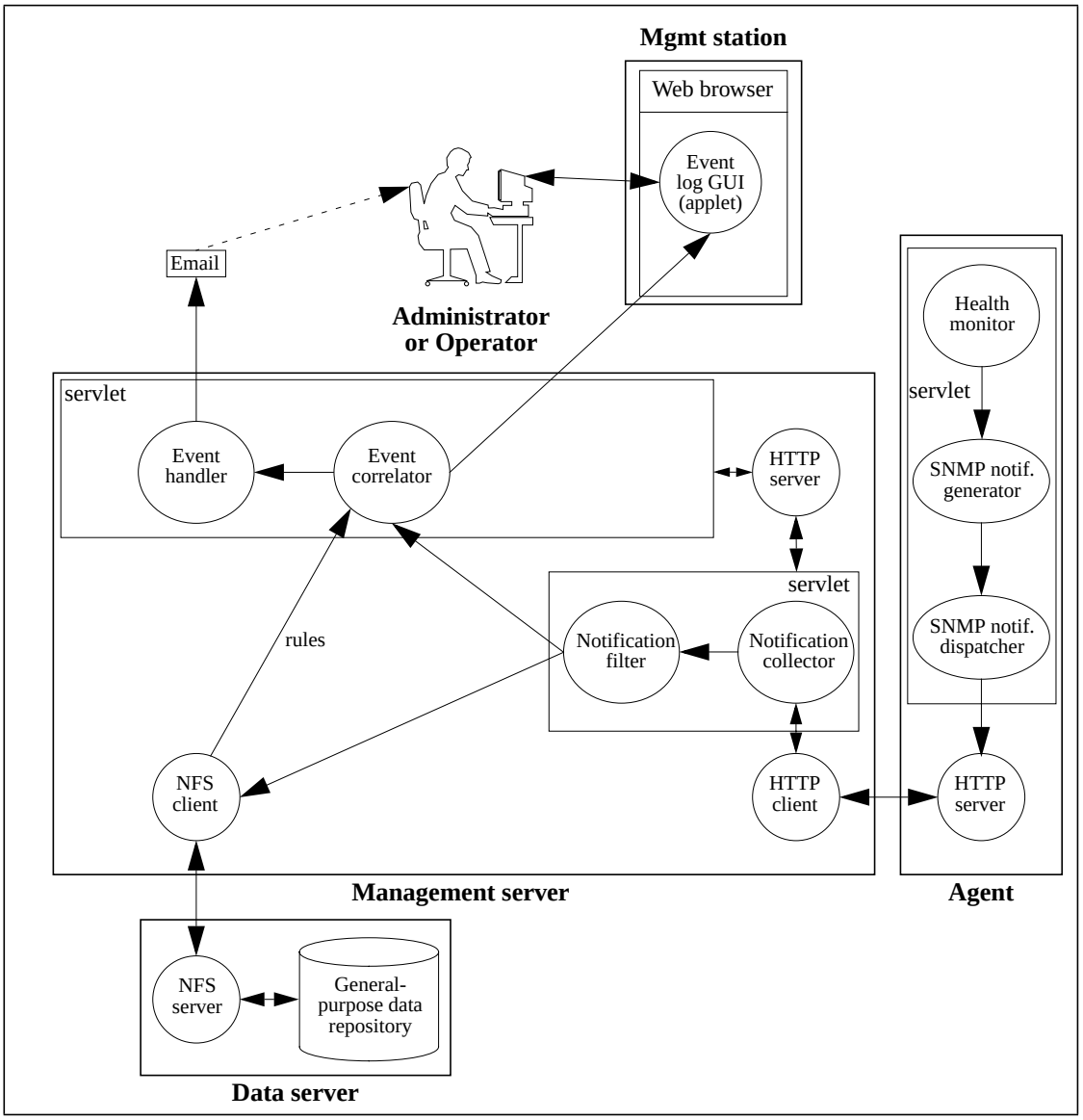

Fig. 44. Push model in JAMAP: notification delivery and event handling

The distribution phase for notification delivery is depicted in Fig. 44, together with event handling within the management server. If we compare this figure with the generic one presented in Chapter 6 for WIMA (Fig. 14, p. 108), the main difference is again the presence of Java servlets in JAMAP. We clearly see the separation between the notification collector servlet, in charge of receiving a specific kind of event coming from agents (notifications), and the event manager servlet, in charge of correlating all types of events (see Section 9.3).

Another difference between this figure and the generic WIMA figure is the absence of sensors in the agent. In Section 9.4.2, we will see that the health monitor implemented in JAMAP is in fact an alarm simulator, which does not use real data coming from sensors.

Finally, as in the previous section, we also use an event log GUI instead of a network-map GUI, and event handlers must use email for notifying the administrator of a problem.

This concludes our description of how JAMAP implements WIMA. In the next three sections, we will investigate the detailed design of the different applets and servlets running on the three tiers of our management architecture: the management station, the management server, and the agent.

9.2 Management Station

The management station is the desktop of the administrator or operator. It can be any machine (a Linux PC, a Windows PC, a Mac, a Unix workstation, etc.) as long as it runs a Web browser and supports Java. Unlike the management server, the management station is not static: the administrator can work on different machines at different times of the day. During the subscription phase, he/she configures the agent via the SNMP MIB data subscription applet and the SNMP notification subscription applet. The rules used by the pushed-data collector and the event manager servlets can be modified at any time via the rule edition applet. Events are displayed by the event log applet.

9.2.1 SNMP MIB data subscription applet

The SNMP MIB data subscription applet communicates directly with the agent (see Fig. 43). It provides the subscription system for regular management. It is also used to retrieve and view data interactively, either once (pull based) or over a longer period of time (push based). Its main tasks are the following:

- browse SNMP MIBs graphically
- select SNMP MIB variables or SNMP tables and retrieve their values once (pull model)
- select SNMP MIB variables and monitor them for a while (text fields, time graphs or tables)
- monitor some computed values (e.g. interface utilization)
- subscribe to SNMP MIB variables or SNMP tables and specify a push frequency (per MIB variable)

Computed values are typically the results of equations parameterized by multiple SNMP MIB variables. We implemented a sort of multiplexer to support them. This kind of simple preprocessing could be delegated to the agent in the future (e.g., with mobile code).

9.2.2 SNMP notification subscription applet

Similarly, the SNMP notification subscription applet also communicates directly with the agent (see Fig. 44). It enables the administrator to set up a filter for notifications at the agent level. SNMP notifications that have not been subscribed to by the manager are silently discarded by the agent.

9.2.3 Rule edition applet

The rule edition applet (see Fig. 41 and Fig. 42) controls the behavior of two objects:

- the pushed-data interpreter object, which lives in the pushed-data collector servlet
- the event correlator object, which lives in the event manager servlet

The administrator can write rules in Java via the applet, or can edit them separately and apply them via the applet. (Java is used here as a portable scripting language.) For instance, an event can be generated by the pushed-data interpreter if the value of an SNMP MIB variable exceeds a given threshold. A typical rule for the event correlator would be that if a system is believed to be down, then all applications running on it should also be down, so events reporting that NFS is not working or that an RDBMS is not working should be discarded.

More complex rules can easily be written. For instance, the pushed-data interpreter can check if the average value of a given SNMP MIB variable increased by 10% or more over the last two hundred push cycles. In fact,

these rules can be arbitrarily complex, as there is no clear-cut distinction between what is in the realm of offline data mining and what should be performed immediately, in pseudo real-time. The trade-off is that the pushed-data interpreter should not be slowed down too much by an excessive amount of rules, otherwise it might not be able to apply all the relevant rules to incoming data between two consecutive push cycles.

9.2.4 Event log applet

The event log applet (see Fig. 43 and Fig. 44) is connected to the management server to receive events. We use it as a debugger, as we do not manage a production network with our platform. This applet displays a simple list of events and manages a blinking light and sound system to grab the operator's attention in case of incoming events. It is intended to remain permanently in a corner of the administrator's and operator's desktop screens. Eventually, it should be complemented by (or replaced with) the network map GUI applet.

9.3 Management Server

The management server runs three servlets: the pushed-data collector, the notification collector, and the event manager. These servlets are not specific to SNMP, and we can easily support other information models through subtyping. In principle, this management server could easily be distributed over multiple machines if need be (e.g., for scalability reasons), as the communication between servlets relies on HTTP and the data server is already a separate machine. For instance, we could run the three servlets on three different machines, and data mining on a fourth. This simple type of distribution, which preserves a centralized management paradigm, was described in Section 6.3.5 (see Fig. 16). But so far, we have only tested JAMAP with a single machine for the management server.

9.3.1 Pushed-data collector servlet

The pushed-data collector servlet consists of three core objects (see Fig. 43), plus a number of instrumentation objects not represented on that figure. The pushed-data collector object connects to the agent upon start-up, and enters an infinite loop where it listens to the socket for incoming data and passes this data unchanged to the pushed-data filter object. If the agent gracefully closes the persistent connection, e.g. in case of a clean reboot, the pushed-data collector immediately reconnects to it so as to ensure a persistent connection. Note that the advanced keepalive schemes described in Section 7.5 are not yet implemented in JAMAP.

The pushed-data filter object controls the flow of incoming data. If it detects that too much traffic is coming in from a given agent (that is, from a given socket), it tells the pushed-data collector object to close permanently the connection to that agent (that is, the collector should not attempt to reconnect to the agent until the administrator explicitly tells it to do so). The rationale here is that a misbehaving agent is either misconfigured, bogus, or under the control of an intruder pursuing a denial of service attack, and that the good health of the management system should be protected against this misbehaving agent. When this happens, the administrator is informed via email.

If the pushed-data filter object is happy with the incoming data, it passes it unchanged to the pushed-data interpreter object. The latter unmarshalls the data and checks, MIB variable by MIB variable, whether it was subscribed to for monitoring, data collection, or both.

In the case of data collection, the SNMP MIB variable is not processed immediately. Instead, it is stored in a persistent repository (an NFS-mounted file system) via a logger object. We assume that an external process will use it afterward to perform some kind of data mining (e.g., it could look for a trend in the variations of the CPU load of an IP router to be able to anticipate when it should be upgraded).

In the case of monitoring, the SNMP MIB variable is processed immediately. The pushed-data interpreter object applies the rules relevant to that agent and that MIB variable. If it notices something important (e.g., a

heartbeat is received from an IP router that was considered down), the pushed-data interpreter object generates an urgent event and sends it via HTTP to the event correlator object living in the event manager servlet. We took special care for the case where the same SNMP MIB variable is used for both monitoring and data collection. The data is then duplicated by the pushed-data interpreter.

A nice feature of our rule system is that rules may be dynamically compiled and loaded in by the servlet. Dynamic class loading is a feature of the Java language. The core API provides a method to instantiate objects from a class by giving its name in the form of a string. The class loader of the JVM searches the class file in the file system, and loads it into the JVM's memory. This enables the servlet to load a class at runtime without knowing its name in advance. Once a class is loaded, it behaves just as any other class. We are limited only by the fact that a class cannot be modified at runtime. This means that if a rule is already registered under a certain class name and that rule is modified by the administrator, another class name must be used for that new version of the rule.

To solve this problem, we implemented a simple technique that consists in postfixing the class name with a release number and incrementing this release number automatically. As a result, the administrator can create, modify, and debug rules dynamically. The drawback is that the memory used by loaded classes (especially those corresponding to the "old" rules) is freed only when the JVM is restarted. The administrator should therefore be careful not to fill up the memory in the rule debugging phase. Clearly, this feature should be used with special care on a production system; but it proved to be particularly useful to us for debugging rules.

9.3.2 Notification collector servlet

As depicted in Fig. 44, the notification collector servlet consists in principle of two core objects: the notification collector and the notification filter. Contrary to pushed data, no interpreter is needed for notifications because we know already what happened: we do not have to work it out.

The notification collector object works exactly as the pushed-data collector object. The notification filter object also works as the pushed-data filter object. In fact, in the current version of JAMAP, the notification collector servlet and the pushed-data collector servlet are one single servlet. This enables us to use a single persistent connection between the agent and the manager for transferring SNMP MIB data and notifications. (Note that this would not be the case if we were to distribute the servlets over several machines or if we wanted to manage different priority levels for notifications, as described in Section 7.4.2.) Notifications received by the pushed-data interpreter object are currently passed on unchanged to the event correlator object living in the event manager servlet, without any further processing.

9.3.3 Event manager servlet

The event manager servlet connects to one or more pushed-data collector servlets (one in the case depicted in Fig. 41 and Fig. 42) and waits for incoming events. Events are processed by the event correlator object. This object performs a simple correlation with regard to the network topology, in order to discard masked events. For instance, if a router is down, all machines accessed across it will appear to be down to the pushed-data interpreter. Based on its knowledge of the network topology (which is hardcoded in the current version of JAMAP), the event correlator is able to keep only those events that cannot be ascribed to the failure of other equipment.

When an event is not discarded by the event correlator object, it is transmitted to the event handler object corresponding to its level of emergency (this emergency level is encapsulated inside the event). Each event handler is coded to interface with a specific notification system (e.g., an email system, a pager, a telephone, a siren, etc.). In our prototype, we only implemented an email-based notification system.

In the future, it would be useful to replace our simple event correlator with a full-blown event correlator written in Java by another research team. This would fit nicely into the "spirit" of component software.

9.4 Agent

The agent runs two servlets: the SNMP MIB data dispatcher and the SNMP notification dispatcher. Both of them are specific to the SNMP information model.

9.4.1 SNMP MIB data dispatcher servlet

The SNMP MIB data dispatcher servlet consists of three core objects (the push scheduler, the MIB data formatter, and the SNMP MIB data dispatcher) plus a number of instrumentation objects not represented on Fig. 43. During the subscription phase, the push scheduler object stores locally the subscription sent by the SNMP MIB data subscription applet (we call it the agent's configuration). Later, during the distribution phase, the push scheduler object uses this configuration to trigger the push cycles. It tells the SNMP MIB data formatter object what SNMP MIB variables should be sent at a given time step. The SNMP MIB data formatter object accesses the in-memory data structures of the SNMP MIBs via some proprietary, tailor-made mechanism (currently, AdventNet's Java classes), formats the SNMP MIB data as a series of {OID, value} pairs, and sends it to the SNMP MIB data dispatcher object. The latter compresses the data with `gzip`, assembles the data in the form of a MIME part, pushes the MIME part through, and sends a MIME separator afterward to indicate that the push cycle is over. In the current version of JAMAP, we can have only one {OID, value} pair per MIME part. In the next version, we should allow for several.

In the future, the SNMP MIB data dispatcher servlet should be able to retrieve the agent's configuration from the data server via the management server. Thus, the agent would not necessarily have to store its configuration in nonvolatile memory—a useful feature for bottom-of-the-range equipment.

9.4.2 SNMP notification dispatcher servlet

The SNMP notification dispatcher servlet consists of two core objects (the SNMP notification generator and the SNMP notification dispatcher) plus a number of instrumentation objects not represented on Fig. 44. During the subscription phase, the SNMP notification generator object stores locally the subscription sent by the SNMP notification subscription applet. In other words, it sets up a filter for SNMP notifications coming in from the health monitor. During the distribution phase, the health monitor checks continuously the health of the agent based on input from sensors. When a problem is detected, the health monitor asynchronously fires an alarm to the SNMP notification generator object in the servlet via some proprietary mechanism. The SNMP notification generator object checks with the filter if this alarm should be discarded. If it was not subscribed to by the manager, the alarm is silently dropped. If it was, the SNMP notification generator object formats it as an `SNMPv2-trap` PDU and sends it to the SNMP notification dispatcher object, which, in turn, wraps it in the form of a MIME part, pushes it to the management server via HTTP, and sends a MIME separator afterward to indicate that this is the end of the notification.

As we do not manage a real-life network with JAMAP, the SNMP notifications that are generated by the health monitor are all simulated. Instead of using real sensors, we use a script that fires, from time to time, one notification taken in a pool of predefined notifications; the selection of this notification is based on a random number generator. As with the previous servlet, the SNMP notification dispatcher servlet should eventually be able to retrieve the agent's SNMP notification filter from the data server via the management server.

9.5 Reusability

In order to reduce our development efforts, we reused a number of existing classes in JAMAP. They are briefly described in this section. This demonstrates, if need be, that reusable component software is not simply a vision!

AdventNet SNMP suite

We used the following classes of the AdventNet SNMP suite [1]:

- the `MibTree` class: a Java bean displaying the SNMP MIB tree as a GUI
- the `MibNode` class: a node of the SNMP MIB tree
- the `SnmpVar` class: represents/encodes an SNMP variable
- the `SnmpTarget` class: a Java bean abstracting an SNMP device
- the `SnmpTable` class: a Java bean representing an SNMP table

HTTPClient

We used the `Util` class of the HTTPClient package written by Tschalär [227]. HTTPClient provides more features than the network classes found in JDK 1.1.7, especially for parsing HTTP headers.

IBM AlphaWorks SMTP

We used the `SMTPConnection` class [104] in the event mailer consumer (event handler) to send email to administrators and operators.

JDK's Java compiler class

We used the `sun.tools.javac.Main` class of Sun Microsystems's JDK Java compiler [217] to implement dynamic compilation in the rule system (described in Section 9.2.3). Note that this class may change between successive versions of the JDK, which hampers the portability of the code of our rule editor.

9.6 Summary

The purpose of developing JAMAP, a WIMA-based research prototype, was to demonstrate the feasibility and simplicity of the core contributions of our work, primarily the use of push, persistent HTTP connections, and MIME multipart. JAMAP is written entirely in Java. It implements the push model to perform regular management (permanent monitoring and data collection for offline analysis) and notification delivery, and the push and pull models for *ad hoc* management (temporary monitoring and troubleshooting). The communication between agents and managers relies on HTTP transfers between Java applets and servlets over persistent TCP connections. The SNMP MIB data is encapsulated in serialized Java objects that are transmitted as MIME parts via HTTP. This data is transparently compressed with `gzip`, which saves network bandwidth without increasing latency too significantly. The manager consists of two parts: the management server, a static machine that runs the servlets, and the management station, which can be any desktop running a Web browser. The other actors involved in a three-tier, client-server architecture are the agent, which supports native SNMP MIBs, and the data repository, which consists of binary files accessed via NFS by the management server.

Chapter 10

How Does Our Solution Compare with Others?

In this chapter, we compare our approach with its main contenders to date for the next management cycle, and highlight the relevance of our architectural and design choices.

This chapter is organized as follows. In Section 10.1, we compare WIMA with the SNMP management architecture, whose replacement was an explicit objective of this work, and demonstrate that we succeeded in solving the problems identified in Chapter 2. In Section 10.2, we compare WIMA to WBEM, identify some similarities, but also some problems in WBEM. An important result is the complementarity of these two approaches. In Section 10.3, we compare WIMA with JMX. We point out the strengths and weaknesses of JMX, and conclude that it can hardly be integrated with WIMA. In Section 10.4, we analyze some known weaknesses in WIMA and explain how to avoid or live with them. Finally, we summarize this chapter in Section 10.5.

10.1 Comparison with SNMP-Based Management

In this section, we first prove that almost all the problems identified in SNMP are solved in WIMA. Then we investigate the complementarity between these two management architectures.

10.1.1 Almost all the problems in SNMP have been solved

If we review the characteristics of SNMP described in Section 2.2, and the problems identified in SNMP-based management in Section 2.4, we see that WIMA is a better solution for the next management cycle:

- Agents are no longer considered equal and “dumb”: some agents can execute scripts transferred via XML, others can simply push MIB variables.
- Polling is no longer used for retrieving regular management data. This improves scalability, network overhead, and latency.
- The data repository is no longer tightly coupled with the management platform. This frees customers from the impediments of peer-to-peer agreements between database and management-platform vendors: they can use any third-party database with any management platform. In particular, they can reuse for NSM the database already installed in their intranet.

- In WIMA, the goal shifts from network to integrated management. WIMA facilitates the integration of different management areas, notably with XML, and does not concentrate solely on the management of network devices.
- WIMA is scalable. Through distribution across a hierarchy of managers, it can cope with the ever growing amount of management data to move about and process at the manager(s). It can also transfer some of the management workload to the agents.
- WIMA makes bulk transfers of management data much more efficient than in the SNMP realm.
- WIMA is immune to the `get-bulk` overshoot effect because it does not use `get-bulk`.
- WIMA is immune to the problems caused by the maximum size of an SNMP message because it does not use the SNMP communication protocol.
- By transferring tables in XML, WIMA makes it possible for smart agents to suppress the "holes" in sparse tables described in Section 2.4.1.3.
- By allowing the agent to send large amounts of data in one bulk, we significantly improve the consistency of large SNMP tables.
- BER encoding is no longer mandatory, so we are no longer exposed to its weak efficiency and its impact on the network overhead.
- Management data can be compressed transparently, which reduces network overhead dramatically without affecting latency too much.
- In WIMA, we use a reliable transport protocol. We do not lose important notifications for silly reasons such as buffer overflows in IP routers.
- In WIMA, we can leverage Web security (SSL, HTTP authentication, etc.) to provide different levels of security. In SNMP, we have either no security (SNMPv1 and v2c) or high security (SNMPv3).
- Firewalls are difficult to cross with SNMP. They were extremely rare when SNMPv1 was devised, and SNMP made no special provisions for crossing them easily, not even SNMPv3. In WIMA, we solved this problem by changing the communication model to accommodate firewall constraints.
- In SNMP, we have no high-level semantics, only instrumentation MIBs. By making it possible to work with other information models, especially CIM, WIMA allows management-application designers to work with high-level semantics. The flexibility of XML is particularly useful for this. The DMTF is currently working on schemata offering high-level semantics.
- The scarcity of protocol primitives in SNMP is addressed in WIMA by abandoning the SNMP communication protocol, by using HTTP and XML instead, and by allowing for feature-rich information models, especially CIM that supports an infinitely large number of operations.
- The SNMP information model is not object oriented, whereas object orientation is now ubiquitous in software engineering. By allowing for the use of CIM, WIMA gives access to an object-oriented information model.
- SNMP's distasteful programming by side effect can be avoided in WIMA: with CIM, clean method invocations are possible.
- In WIMA, component- and object-oriented management platforms are more modular than SNMP-based management platforms, and competition should make them less expensive.
- To vendors, the possibility to have a single device- or vendor-specific management GUI for all management platforms is an immense source of savings. This should contribute to making the cost of embedded management software remain low, and therefore decrease the software bill for customers.
- The time-to-market of vendor-specific management GUIs is brought down to zero in WIMA, a remarkable improvement over SNMP where it typically takes several months, and sometimes infinitely longer.
- Start-ups are on a par with large equipment vendors regarding management: they are no longer left aside because of the high entrance cost in the management-platform market.
- MIB versioning is solved in WIMA by embedding the vendor-specific management GUI in the agent.
- Domain-specific expertise is no longer needed: WIMA relies on standard Web technologies, well known in software engineering today.
- Finally, we mentioned that SNMP evolves too slowly. By allowing WIMA to cope with any information model, we make it possible for new entrants (e.g., the DMTF with CIM) to offer quicker solutions. If new SNMP MIBs are too slow to come, we can use CIM schemata instead.

Among all the problems identified in Chapter 2, only three of them are not solved in WIMA:

- Because we decided to make it possible to work with legacy SNMP MIBs, the verbosity of the OID naming scheme in SNMP is not solved. This is partially addressed by the possibility to compress management data in transit.
- We still need a dedicated machine for the management server.
- The possibility to move the software of the management server from one machine to another is not guaranteed by WIMA. It can be achieved, e.g. by coding the components in Java. But there is a trade-off to be made between portability and efficiency.

10.1.2 Complementarities between WIMA and SNMP

The complementarity between WIMA and SNMP is primarily guaranteed by the fact that WIMA can cope with any information model, and in particular SNMP MIBs. It is also possible to use BER encoding inside HTTP messages to interface easily with existing SNMP-based management platforms.

10.2 Comparison with WBEM

In this section, we compare WIMA with WBEM. We first show some similarities in these two approaches. Then, we unveil three problems in WBEM. Finally, we investigate the complementarity between these two management architectures.

10.2.1 Similarities: HTTP and XML

Although they do not share the same communication model, WIMA and WBEM both use HTTP for agent-manager communication, and XML for representing management data inside HTTP messages. In WBEM, the use of XML is mandatory; in WIMA, it is simply recommended.

It should be noted that when the author made his design decision to use HTTP and XML in WIMA, shortly after the W3C had released XML, he was not aware that the WBEM Consortium had already abandoned HMMP in favor of HTTP and XML. This similarity is thus the result of sheer coincidence, as opposed to cross fertilization.

10.2.2 Problems with WBEM

We identified three problems in WBEM: the lack of an organizational model, the use of HTTP extension headers, and the possible link with DEN.

10.2.2.1 WBEM lacks an organizational model

There is one deficiency in WBEM: it specifies information and communication models, but it lacks an organizational model. There is no recommendation as to how managers and agents should interact and share the management-application workload. Issues such as the manager-agent paradigm, mobile code, or the push and pull models have not yet been addressed. As a result, when people eventually build WBEM-based management applications, they will probably map what they are familiar with onto WBEM. In the IP world, and especially in network management, this means that they will build their management application around the pull model, despite all the problems unveiled in this work.

In WIMA, we clearly specify when the push model should be used, and when the pull model is preferred.

10.2.2.2 HTTP extension headers

The reliance of WBEM on HTTP extension headers (see Section 5.5.1) causes three problems: the reliance on a controversial extension scheme for HTTP/1.1, the pseudo support for HTTP/1.0, and the compliance deadlock for embedded HTTP servers.

Controversial extension scheme for HTTP/1.1

HTTP/1.0 and HTTP/1.1 both define a large set of HTTP header fields, often called *HTTP headers* for short. None of them allows for domain-specific HTTP headers. In other words, to claim compliance with HTTP/1.0 or HTTP/1.1, an application must not define and use its own HTTP headers. Over time, this has caused some dissatisfaction in the HTTP community, because many working groups are currently building architectures on top of HTTP, and many of them want to add domain-specific HTTP headers to the standard HTTP headers. A solution to this problem was recently proposed in RFC 2774 [155]: HTTP extension headers. This RFC is a proposed extension to HTTP/1.1. It specifies a set of conventions that allow applications to define and use domain-specific headers with HTTP/1.1.

The problem with this approach is that it has caused a lot of controversy at the W3C and IETF. This is testified by the unusual beginning of RFC 2774:

> "IESG Note
>
> This document was originally requested for Proposed Standard status. However, due to mixed reviews during Last Call and within the HTTP working group, it is being published as an Experimental document. This is not necessarily an indication of technical flaws in the document; rather, there is a more general concern about whether this document actually represents community consensus regarding the evolution of HTTP. Additional study and discussion are needed before this can be determined." [155, p. 1]

CIM operations over HTTP depend on HTTP extension headers to work (see [68, Sections 3.1, 3.3.4, 3.3.5, 3.3.6, 3.3.7, and 3.3.8]). In our view, it seems unwise to base an important building block of the WBEM management architecture on a technology whose future is so uncertain.

HTTP extension headers break HTTP/1.0

RFC 2774 makes it clear that the use of HTTP extension headers demands HTTP/1.1:

> "The proposal uses features in HTTP/1.1 but is compatible with HTTP/1.0 applications in such a way that extended applications can coexist with existing HTTP applications. Applications implementing this proposal MUST be based on HTTP/1.1 (or later versions of HTTP)." [155, p. 3]

As a result, CIM operations over HTTP do not work with HTTP/1.0:

- If we abide by the HTTP/1.0 specification, we cannot use domain-specific HTTP headers, so we cannot define CIM extension headers, thus CIM operations cannot be encapsulated in HTTP.
- If we do use CIM extension headers, we use domain-specific HTTP headers, hence we break HTTP/1.0.

This is in contradiction with the claimed support for HTTP/1.0 in the specification for CIM operations over HTTP:

> "In recognition of the large installed base of HTTP/1.0 systems, the encapsulation is designed to support both HTTP/1.0 and HTTP/1.1." [68, p. 5]
>
> "It is RECOMMENDED that CIM clients and CIM servers support HTTP/1.1. CIM clients and servers MAY support HTTP/1.0 instead." [68, p. 44]

This issue was recently brought before the DMTF WBEM Interoperability Working Group by the author. It is serious because the vast majority of the embedded HTTP servers deployed to date are based on HTTP/1.0. As a result, CIM operations over HTTP do not work with most deployed equipment, unless applications deliberately break the HTTP/1.0 specification...

Compliance deadlock

Another problem is the compliance deadlock. When customers purchase a new piece of equipment, their request for bids can require the support for an embedded HTTP server. Arguably, they can even require an HTTP/1.1 server, although few vendors, according to our sources, give customers the choice between HTTP/1.0 and HTTP/1.1. But how can they possibly require an HTTP/1.1 server modified with WBEM's HTTP extension headers, when HTTP extensions are simply an experimental RFC? This is completely unrealistic. Moreover, if other working groups define other HTTP extension headers for other purposes (e.g., for dynamic service provision), who will ensure that all these extensions are not mutually exclusive? And how will equipment vendors keep up with the definition of new extension schemes? As it is currently specified, WBEM's reliance on HTTP extension headers is a time bomb, in our view.

10.2.2.3 DEN

The third problem that we identified in WBEM is the growing importance given to DEN by the DMTF. DEN specifies one way of storing data. Arguably, it might be the best technology to date for storing the management data necessary for automated configuration (see Sun Microsystems' Jini and Microsoft's Universal Plug and Play). But recent DMTF newsletters lead us to believe that some people at the DMTF contemplate the prospect of including DEN in WBEM. In our view, this would be a mistake. WBEM should not attempt to standardize the way management data is stored in data repositories. DEN is simply a means to an end: the end is integrated management.

10.2.3 Complementarities between WIMA and WBEM

There is some scope for the integration of WIMA and WBEM. To begin with, WIMA's organizational model could be adopted by WBEM, especially the push model for transferring regular management data. The use of MIME multipart and the encapsulation of XML documents in MIME parts could also be adopted by WBEM. Note that this integration would require the definition of a new scheme for encapsulating CIM operations in HTTP messages, as the use of HTTP extension headers is incompatible with WIMA. We are currently investigating ways to replace HTTP extension headers with XML metadata inside the MIME parts. Further study of the integration of WIMA and WBEM is left for future work.

10.3 Comparison with JMX

In this section, we compare WIMA with JMX and investigate complementarities between these two management architectures.

Similarities: components

The main similarity between WIMA and JMX is the reliance on component software on the manager side. Because of their complexity, management servers are very good candidates for component software. Agents can also be implemented with components, but most of the time, their complexity does not require it, and efficiency demands the faster execution of C or C++ binaries.

The my-middleware-is-better-than-yours syndrome: a killer

JMX fundamentally relies on the Java middleware to be supported by all agents and managers worldwide. In Section 4.4, we explained that this approach is flawed in our view, because it is an instance of the *my-middleware-is-better-than-yours* syndrome. Because it seems highly unlikely that one middleware will eventually win the middleware war, we believe that it would be unwise to base the management architecture of the next management cycle on a middleware that may be rejected by the market in a couple of years. In comparison, WIMA is far less risky.

Note that since the split of Java between J2EE (EJBs for managers), J2SE (JavaBeans for top- and middle-of-the-range agents), and J2ME (components for consumer electronics and embedded devices), it is not clear how a single management architecture can live with heterogeneous components. What happens if a J2EE-based manager uses Java RMI to manage a J2ME-based agent? What is called *Java RMI* in these two worlds is fairly different. This problem is not yet addressed in JMX, to the best of our knowledge.

JMX and FMA

As we mentioned in Section 5.6, JMX has so far focused on the agent side while FMA concentrated on the manager side. JMX uses MBeans. FMA has migrated from domain-specific FederatedBeans to standard JavaBeans and EJBs. Consequently, these two architectures are currently incompatible. We still need an integrated management architecture that will cover both the agent and the manager sides in distributed Java-based management. This, too, is an argument for not selecting JMX for the next management cycle.

Complementarities between WIMA and JMX

The main complementarity that we see between WIMA and JMX is the component-oriented management server, once JMX has specified the manager side. Whether the management server is implemented in Java is transparent to WIMA, and the component-software approach adopted in JMX is consistent with our design decisions in WIMA. The agent side could also be implemented in Java, although the need for component software is less obvious on an agent. Apart from that, WIMA and JMX offer little scope for integration.

10.4 Known Problems with WIMA

WIMA is not immune to problems. We identified three: the reliability of new software, the need for management software integrators, and the need for clock synchronization. We describe them in this section and show that none is fatal in the long term.

Reliability of new software

The main problem in WIMA is the reliability of new management platforms based on COTS components and object-oriented frameworks. To put it simply: new means buggy. Years of debugging and real-life testing have already gone into all the major SNMP-based management platforms, which gives a lot of confidence to potential buyers. Component-oriented software is comparatively new, so the solution that we propose in WIMA is more prone to errors. On the other hand, component software is all about reuse, so component-based management platforms should supposedly take less time to be debugged if the same component is reused in many different contexts.

Note that all possible solutions for the next management cycle are exposed to this problem, except one: SNMP-based management. If we do not change anything, we take no risks, but we have to learn to live with the same problems for many years.

Need for management software integrators

The second source of concern in WIMA is the need for management-software integrators. As we mentioned in Section 6.2.3, components are frequently developed by different vendors in the component-software market. In order to be shielded from liability issues when two components from different vendors do not interact as they are supposed to, most customers should prefer to buy component software from software integrators. These integrators test that the components work fine together, they take care of all legal aspects in case of litigation, and they can also transparently replace a family of components with another if a component is no longer supported (e.g., due to repeated interoperability problems, legal issues, or bankruptcy).

The concerns with software integrators are twofold. First, they have to make some profit, so they reduce the cost savings compared with SNMP-based management platforms. Second, true competition only exists when a customer is ready to visit many vendor booths at a trade show. By going through a software integrator, high are the risks of seeing a reincarnation of the dreaded concept of a “preferred partner”... and of going back to equally dreaded peer-to-peer agreements between vendors. The good news is, because the business of software integration is not overly complex, we can expect many companies to come into this market, thereby increasing competition and the chances for a customer to find a good integrator. The other good news is that, at any point in time, a customer can decide that he/she now knows enough of the component-software market to free him/herself from any integrator.

Synchronization of the clocks

The last issue in WIMA is clock synchronization. If all the managers and agents do not regularly synchronize their internal clocks, some of these clocks will significantly drift apart over time. This is not a problem for monitoring, because it does not matter, when an agent is configured to send a heartbeat to its manager every 5 minutes, whether the manager receives it every 299 seconds or every 301. But it can be a problem for data collection, because the data repository often stores the values for a given MIB variable in the form of a time series. Without synchronization, the manager could receive too many or too few push cycles per day. In this case, to build consistent time series for all MIB variables, some data would have to be discarded arbitrarily, and others would have to be interpolated at a random cycle number. Clearly, this is undesirable. And there is a simple solution: loose synchronization, typically once per day or once per hour.

In WIMA-based management, it is therefore recommended to synchronize the clocks of all the managers and agents on a regular basis. There are many, well-known ways of achieving loose synchronization. Agents and managers can receive clock updates via radio waves. Or they can rely on the use of a protocol like NTP (Network Time Protocol). These two solutions can also be combined: the management server can update its clock with an absolute time received via radio waves, and it can in turn update the agents via NTP. Alternatively, the manager can exchange a few synchronization packets with all the agents in its domain, with an *ad hoc* protocol. Another solution is to make it mandatory for an agent to include a timestamp in each MIME part: we simply have to specify the format of this timestamp in WIMA. In all these cases, the synchronization overhead is negligible compared to the network and CPU savings induced by going from pull to push.

Note that this synchronization problem is caused by the use of the push model: the agent’s clock decides when it is time for the next push cycle, and the manager’s clock may disagree with it. With the pull model, the manager’s clock decides when it is time for the next pull cycle, so time series always get the right number of entries (unless some data does not come in, e.g. in the case of network congestion, in which case it is possible to store an error code such as `notAvailable` for a specific entry, as opposed to a random entry).

10.5 Summary

In this chapter, we showed that WIMA compares very well with the other approaches suggested thus far for the next management cycle. In Section 10.1, we compared WIMA to SNMP-based management and concluded that almost all of the problems identified in SNMP have been solved in WIMA. In Section 10.2 and Section 10.3, we compared WIMA with WBEM and JMX, two standardization efforts begun by industrial consortia at a time when this thesis work was well under way. We identified some problems in these two approaches, but also some complementarities with WIMA. Finally, in Section 10.4, we described some known problems in WIMA and suggested ways to solve them.

Chapter 11

CONCLUSION

In this last chapter, we summarize the main contributions of this Ph.D. work and give some directions for future work.

11.1 Summary

In this dissertation, we have proposed a new architecture for network and systems management in the IP world: WIMA (Web-based Integrated Management Architecture). WIMA is destined to replace SNMP in the next management cycle. Its primary achievement is that it solves almost all the problems that we identified in SNMP-based management (that is, in the solution currently adopted by the IP world). WIMA is also simple to implement and does not rely on unrealistic assumptions such as "All IP network devices and systems worldwide must support the same object-oriented middleware". In this section, we summarize our technical contributions, the relevance of our work, and what is needed to migrate from SNMP to WIMA.

Technical contributions

The core contributions of this Ph.D. work are threefold. First, we advocate that regular management data should be pushed by agents, not pulled by managers. This leads us to adopt a push-based organizational model for regular management in WIMA. Second, we argue that (i) standard Web technologies should be used instead of the domain-specific SNMP, and (ii) management data should be reliably transmitted across persistent TCP connections to avoid the loss of important data. WIMA's communication model is based on persistent HTTP connections between the manager and the agent (or between the mid- and top-level managers in distributed management). Within these connections, HTTP messages are structured with MIME multipart. Each MIME part can be an XML document, a binary file with BER-encoded data, a textual file with data encoded in plain strings, etc. MIME parts can be transparently compressed, which reduces network overhead significantly. By creating connections from the manager side, we facilitate crossing firewalls. Our communication model is totally independent of the information model, which allows us to transfer SNMP data, CIM data, etc. Third, we recommend the use of XML for (i) distributing management across a hierarchy of managers (weakly distributed hierarchical management), (ii) integrating management, (iii) dealing with multiple information models (especially SNMP MIBs and CIM schemata), and (iv) supporting high-level semantics.

This dissertation also includes four secondary contributions. First, two taxonomies classify all the management paradigms that have been proposed to date, and help administrators select a paradigm or technology well suited to their needs. The simple taxonomy is based on a single criterion (the organizational model of the management architecture), while the enhanced taxonomy is based on four criteria (the delegation granularity, the semantic richness of the information model, the degree of automation of management, and the degree of specification of a task). Second, our detailed analysis of the problems with SNMP-based management substantiates a claim often made but rarely substantiated: SNMP-based management is too simple, it will not suffice in the near future. Many assumptions made in the late 1980s, when SNMPv1 was devised, are no longer valid today. So many things should be changed, in the management architecture and the communication protocol, that it is easier to start afresh with a brand-new management architecture. Third, our design innovations are validated by a research prototype, the JAva MAnagement Platform (JAMAP), which demonstrates their simplicity and feasibility. Finally, our state of the art gives a useful overview of the vast and expanding area now known as Web-based management.

Relevance of our work

The problem that we solved with WIMA is not purely academic: it is a real concern to the management industry. As we saw in Chapter 3, many proposals have been made since the mid-1990s to address the problems experienced by SNMP-based management, and most notably scalability. Since the late 1990s, two industrial consortia, the Distributed Management Task Force (DMTF) and the Java Community (centered on Sun Microsystems), have dedicated considerable manpower to solving this problem. So far, CIM and Java-based management are their main deliverables.

The results exhibited by WIMA are significant, too. If we compare it with SNMP-based management (see Section 10.1), we see that almost all the problems identified in SNMP-based management are solved in WIMA. These are not purists' conundrums, these are real-life problems: firewalls, distribution, network overhead, cost, etc. If we compare WIMA with WBEM, we see that it complements the information modeling work of the DMTF and addresses some issues related to the use of HTTP extensions for firewalls. Finally, if we compare WIMA with Java-based management, we see that by using a more flexible design, it is not exposed to the *my-middleware-is-better-than-yours* syndrome.

The third reason that leads us to believe that this Ph.D. work is relevant is the excellent feedback that we received, in the networking industry, from large vendors (e.g., Lucent Technologies and Nortel Networks) and small vendors (e.g., Lightning). The service industry also showed notable interest in our activities, including AT&T and Swisscom.

What does it take to migrate from SNMP- to WIMA-based management?

When a new management architecture comes in, the market always faces a chicken-and-egg situation: equipment vendors are willing to support it if management-platform vendors already do, and *vice versa*. Because writing a new manager is a lot more work than writing a new agent, we believe that the way out of this deadlock is to convince equipment vendors to support WIMA in their network devices and systems. As soon as a number of agents have been shipped with embedded support for WIMA, some start-ups will develop WIMA-compliant management platforms to enter the lucrative management market.

What do equipment vendors need to do to support WIMA? They must embed four software components in their network devices and systems: an HTTP server (we saw that this is often already the case), a push system, a scheduler, and one or several vendor-specific management GUIs. Typically, this software would be embedded in EPROM. Writing the push and scheduling systems is simple and inexpensive. Reprogramming existing management GUIs as Java applets is probably the most costly part, but the potential gains are enormous (write once, run anywhere... if careful enough!).

Once WIMA-compliant agents are available on the market, the second step of the migration process is for management-platform vendors to develop professional-grade component software for management servers. When different components are developed by different vendors, we need software integrators to shield the customers from inter-vendor liability mazes ("It does not work, whose fault is it?"). The current explosion of the Web-based management market, with start-ups flocking in, leads us to believe that we could soon have a mature market of WIMA-compliant management servers—in less than a year. Note that the risk factor for these vendors is lower with WIMA than with most alternatives: object-oriented middleware, mobile code, multi-agent systems, etc.

The third step toward WIMA-based management is for administrators to loosely synchronize the clocks of the managers and agents. There are simple solutions to achieve this, e.g. the Network Time Protocol (NTP). This type of synchronization is already routinely performed in many networks worldwide.

11.2 Directions for Future Work

Many areas could be investigated as a follow-up to this Ph.D. work. We propose five directions for future research work: wireless networks, IP telephony, integrated management, SNMP-to-XML mapping, and the integration of SNMP MIBs and CIM schemata.

WIMA for wireless networks

The work presented herein focuses on fixed IP networks and systems. Preliminary work suggests that WIMA can also cope with IP wireless networks and systems. For instance, push technologies are promising to manage mobile phones and Web-enabled handheld devices, because these devices can remain inaccessible by the manager for extended periods of time under normal circumstances. By using push instead of pull technologies, we prevent the manager from performing many retries when, for instance, the agent is voluntarily switched off, or when it is in an area that is not covered by any antennae.

WIMA for IP telephony

Another worthwhile research area is the suitability of WIMA in QoS-driven networks, and especially in IP telephony. The challenges are great because the telecom world is used to notification-driven management, with smart agents, whereas WIMA is particularly efficient for transfering bulks of management data from the agent to the manager, to let the manager do most of the management application. Still, WIMA can also deal with smart agents, and it would be interesting to compare its efficiency with that of other management architectures. Signaling, for instance, has not yet been investigated in the context of WIMA.

WIMA for integrated management

Although we mentioned several times the advantages of WIMA in integrated management at large, only network and systems management were studied in detail during this Ph.D. work. We already saw that application management is very similar to systems management, so we expect the integration of application management to be straightforward. Early work also suggests that WIMA can easily be extended to policy management, especially when we use XML to represent management data. To an XML document, it is transparent whether the semantics of the management data is high (for policy management) or low (for network and systems management). Service management still remains to be investigated, but the possibility in WIMA to transfer mobile code within XML documents should prove to be useful for dynamic service provision.

SNMP-to-XML mapping: model or metamodel level?

In Section 8.2, we explained the difference between model- and metamodel-level mappings. An important research area for integrating SNMP-compliant agents and managers with their CIM counterparts is whether the SNMP-to-XML mapping should be performed at the model level (that is, one XML document per SNMP MIB), or at the metamodel level (that is, one XML document for the entire SNMP information model).

Integration of SNMP MIBs and CIM schemata

A fifth interesting research area is the integration of the SNMP and CIM information models. What are the issues? What semantics do we lose when we translate CIM schemata into SNMP MIBs, and *vice versa*? A few years ago, IIMC and JIDM studied information-model integration for SNMP, OSI, and CORBA (see Section 3.1.6.2). It would be very useful to extend this work to CIM, which is backed by many vendors and seems likely to gradually complement, or perhaps even replace, the SNMP information model in the next management cycle.

LIST OF ACRONYMS

ACK	ACKnowledgment
API	Application Programming Interface
ARP	Address Resolution Protocol
ASN.1	Abstract Syntax Notation 1
ATM	Asynchronous Transfer Mode
A-TRT	Agent's TCP Retransmission Timer
A-TRTO	Agent's TCP Retransmission TimeOut
AWT	Abstract Window Toolkit
BDI	Belief Desire Intention
BER	Basic Encoding Rules
BNF	Backus-Naur Form
BOF	Birds Of a Feather
BSD	Berkeley Software Distribution
CD-ROM	Compact Disk - Read-Only Memory
CER	Canonical Encoding Rules
CGI	Common Gateway Interface
CIM	Common Information Model
CIMOM	Common Information Model Object Manager
CLHS	Component Launched by an HTTP Server
CLI	Command-Line Interface
CLNS	ConnectionLess Network Service
CMIP	Common Management Information Protocol
CMIS	Common Management Information Service
CMU	Carnegie Mellon University
COD	Code On Demand
CONS	Connection-Oriented Network Service
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DAI	Distributed Artificial Intelligence
DAP	Distributed Application Performance

DCOM	Distributed Component Object Model
DEN	Directory-Enabled Networks
DER	Distinguished Encoding Rules
DHCP	Dynamic Host Configuration Protocol
DISMAN	DIStributed MANagement
DMI	Desktop Management Interface
DMTF	Distributed Management Task Force
DoS	Denial of Service
DPE	Distributed Processing Environment
EDI	Electronic Data Interchange
EJB	Enterprise JavaBean
EPROM	Erasable Programmable Read-Only Memory
FCAPS	Fault, Configuration, Accounting, Performance, and Security
FDDI	Fiber Distributed Data Interface
FIPA	Foundation for Intelligent Physical Agents
FMA	Federated Management Architecture
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
GDMO	Guidelines for the Definition of Managed Objects
GIF	Graphics Interchange Format
GRM	General Relationship Model
GUI	Graphical User Interface
HMMP	HyperMedia Management Protocol
HMMS	HyperMedia Management Schema
HMOM	HyperMedia Object Manager
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IDL	Interface Definition Language
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
IIMC	ISO-Internet Management Coexistence
IIOP	Internet Inter-ORB Protocol
I/O	Input/Output
IOS	Internetwork Operating System
IRTF	Internet Research Task Force
IP	Internet Protocol
IPSec	IP Security
ISO	International Organization for Standardization
ISP	Internet Service Provider
ITU-T	International Telecommunication Union - Telecommunication standardization sector
J2EE	Java 2 Enterprise Edition
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition

JAMAP	JAva MAnagement Platform
JDBC	Java DataBase Connectivity
JDK	Java Development Kit
JIDM	Joint Inter-Domain Management
JMAPI	Java Management Application Programming Interface
JMX	Java Management eXtensions
JNI	Java Native Interface
JVM	Java Virtual Machine
KQML	Knowledge Query and Manipulation Language
LAN	Local-Area Network
LDAP	Lightweight Directory Access Protocol
LWER	LightWeight Encoding Rules
M2M	Manager to Manager
MAC	Medium Access Control
MAS	Multi-Agent System
MbD	Management by Delegation
MBean	Management Bean
MCS	Mobile-Code System
MIB	Management Information Base
MIME	Multipurpose Internet Mail Extensions
MIT	Massachusetts Institute of Technology
MMO	Memory-Management Overhead
MOF	Managed Object Format
MRTG	Multi-Router Traffic Grapher
MSS	Maximum Segment Size
M-task	macrotask
μ-task	microtask
M-TKT	Manager’s TCP Keepalive Timer
M-TKTO	Manager’s TCP Keepalive TimeOut
MTU	Maximum Transmission Unit
NFS	Network File System
NIM	Network Information Model
NIS	Network Information Service
NMF	Network Management Forum
NOC	Network Operations Center
NSM	Network and Systems Management
NTP	Network Time Protocol
ODBC	Open DataBase Connectivity
ODMA	Open Distributed Management Architecture
OID	Object IDentifier
ODP	Open Distributed Processing
OLE	Object Linking and Embedding
OMG	Object Management Group
OO agent	Object-Oriented agent
OODBMS	Object-Oriented DataBase Management System

ORB	Object Request Broker
OSI	Open Systems Interconnection
PC	Personal Computer
PDU	Packet Data Unit
PER	Packed Encoding Rules
POS-EWS	POStech Embedded Web Server
QoS	Quality of Service
RDBMS	Relational DataBase Management System
REV	Remote EValuation
RFC	Request For Comments
RMI	Remote Method Invocation
RM-ODP	Reference Model - Open Distributed Processing
RMON	Remote MONitoring
RPC	Remote Procedure Call
RST	Reset
RTO	Retransmission TimeOut
RTP	Real-time Transport Protocol
SCTP	Stream Control Transmission Protocol
SLA	Service-Level Agreement
SME	Small or Midsize Enterprise
SMI	Structure of Management Information
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
SS7	Signaling System No. 7
SSL	Secure Sockets Layer
SYN	SYNchronize
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TINA	Telecommunications Information Networking Architecture
TMF	TeleManagement Forum
TMN	Telecommunications Management Network
TOS	Type Of Service
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VACM	View-based Access Control Model
VPN	Virtual Private Network
WAIS	Wide-Area Information Servers
WAN	Wide-Area Network
WbASM	Web-based ATM-Switch Management
WBEM	Web-Based Enterprise Management
WIMA	Web-based Integrated Management Architecture
WIMA-CM	Web-based Integrated Management Architecture - Communication Model
WIMA-OM	Web-based Integrated Management Architecture - Organizational Model
XML	eXtensible Markup Language
xmlCIM	CIM-to-XML mapping

REFERENCES

1. AdventNet. *SNMP Package*. March 1999. Available at <http://www.adventnet.com/products.html>.
2. Agent Society. *Home Page*. Available at <http://www.agent.org/>. April 2000.
3. S. Aidarous and T. Plevyak. “Chapter 1. Principles of Network Management”. In S. Aidarous and T. Plevyak (Eds.), *Telecommunications Network Management into the 21st Century: Techniques, Standards , Technologies and Applications*, pp. 1–18. IEEE Press, New York, NY, USA, 1994.
4. D.S. Alexander, W.A. Arbaugh, M.W. Hicks, P. Kakkar, A.D. Keromytis, J.T. Moore, C.A. Gunter, S.M. Nettles, and J.M. Smith. “The SwitchWare active network architecture”. *IEEE Network*, 12(3):29–36, 1998.
5. B. Alpers and H. Plansky. “Concepts and Application of Policy-Based Management”. In A.S. Sethi, Y. Raynaud, and F. Faure-Vincent (Eds.), *Integrated Network Management IV. Proc. 4th IFIP/IEEE International Symposium on Integrated Network Management (ISINM’95), Santa Barbara, CA, USA, May 1995*, pp. 57–68. Chapman & Hall, London, UK, 1995.
6. S.W. Ambler. *Building Object Applications That Work*. Cambridge University Press, Cambridge, UK, 1998.
7. N. Anerousis. “Scalable Management Services Using Java and the World Wide Web”. In A.S. Sethi (Ed.), *Proc. 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM’98), Newark, DE, USA, October 1998*, pp. 79–90.
8. N. Anerousis. “An Information Model for Generating Computed Views of Management Information”. In A.S. Sethi (Ed.), *Proc. 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM’98), Newark, DE, USA, October 1998*, pp. 169–180.
9. N. Anerousis. “A Distributed Computing Environment for Building Scalable Management Services”. In M. Sloman, S. Mazumdar, and E. Lupu (Eds.), *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM’99), Boston, MA, USA, May 1999*, pp. 547–562. IEEE, Piscataway, NJ, USA, 1999.
10. Apache Software Foundation. *The Apache Project*. Available at <http://www.apache.org/httpd.html>. May 2000.
11. T.K. Apostolopoulos and V.C. Daskalou. “Temporal Network Management Information Model and Services”. *Journal of Network and Systems Management*, 6(3):245–265, 1998.
12. M. Armstrong. *A Handbook of Personnel Management Practice*. 4th edition. Kogan Page, London, UK, 1991.
13. C. Atkinson. “Meta-Modeling for Distributed Object Environments”. In *Proc. 1st International Enterprise Distributed Object Computing Conference (EDOC ’97), Gold Coast, Australia, October, 1997*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1997.
14. ATM Forum. *SNMP M4 Network Element View MIB*. Revision AF-NM-0095.001. July 1998.
15. M. Baldi, S. Gai, and G.P. Picco. “Exploiting Code Mobility in Decentralized and Flexible Network Management”. In K. Rothermel and R. Popescu-Zeletin (Eds.), *Proc. 1st International Workshop on Mobile Agents (MA’97), Berlin, Germany, April 1997*. LNCS 1219:13–26, Springer, Berlin, Germany, 1997.
16. M. Baldi and G.P. Picco. “Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications”. In R. Kemmerer and K. Futatsugi (Eds.), *Proc. 20th International Conference on Software Engineering (ICSE’98), Kyoto, Japan, April 1998*, pp. 146–155. IEEE, Piscataway, NJ, USA, 1998.
17. G Banga, J.C. Mogul, and P. Druschel. “A Scalable and Explicit Event Delivery Mechanism for UNIX”. In *Proc. 1999 USENIX Annual Technical Conference (Usenix’99)*, Monterey, CA, USA, June 1999.
18. F. Barillaud, L. Deri, and M. Feridun. “Network Management using Internet Technologies”. In A. Lazar, R. Saracco, and R. Stadler (Eds.), *Integrated Network Management V. Proc. 5th IFIP/IEEE International Symposium on Integrated Network Management (IM’97), San Diego, CA, USA, May 1997*, pp. 61–70. Chapman & Hall, London, UK, 1997.
19. W.J. Barr, T. Boyd, and Y. Inoue. “The TINA Initiative”. *IEEE Communications Magazine*, 31(3):70–76, 1993.
20. H. Berndt, T. Hamada, and P. Graubmann. “TINA: Its Achievements and Its Future Directions”. *IEEE Communications Surveys & Tutorials*, 3(1):2–16, 2000.

21. T. Berners-Lee, R. Fielding, and H. Frystyk (Eds.). *RFC 1945. Hypertext Transfer Protocol -- HTTP/1.0*. IETF, May 1996.
22. T. Berners-Lee, R. Fielding, and L. Masinter (Eds.). *RFC 2396. Uniform Resource Identifiers (URI): Generic Syntax*. IETF, August 1998.
23. L. Bernstein and C.M. Yuhas. “Truce in Protocol Wars”. *Journal of Network and Systems Management*, 1(2):103–105, 1993.
24. A. Bieszczad and B. Pagurek. “Towards Plug-and-Play Networks with Mobile Code”. In *Proc. International Conference for Computer Communications (ICCC’97), November 1997, Cannes, France*.
25. A. Bieszczad, T. White, and B. Pagurek. “Mobile Agents for Network Management”. *IEEE Communications Surveys*, 1(1):2–9, 1998.
26. BMC Software, Cisco, Compaq, Intel, and Microsoft. *Industry Leaders Propose Web-Based Enterprise Management Standards Effort*. Press Release, July 1996. Available at <http://www.microsoft.com/mscorp/presspass/press/1996/jul96/WEBMAN~1.htm>.
27. R. Booth. *XML As a Representation for Management Information—A White Paper*. Draft. Microsoft, May 1998.
28. J. Bosak and T. Bray. “XML and the Second-Generation Web”. *Scientific American*, May 1999.
29. R. Boutaba. *Une architecture et une plate-forme distribuée orientée objet pour la gestion intégrée de réseaux et de systèmes* (in French). Ph.D. thesis, Pierre & Marie Curie University, Paris, France, March 1994.
30. L. Bovet. *The Push Model in a Java-Based Network Management Application*. M.S. thesis, Computer Science Dept., EPFL, Lausanne, Switzerland, March 1999.
31. R. Braden (Ed.). *RFC* 1122. *Requirements for Internet Hosts—Communication Layers*. IETF, October 1989.
32. N. Bradley. *The XML Companion*. Addison-Wesley, Harlow, UK, 1998.
33. M. Breugst and T. Magedanz. “Mobile Agents—Enabling Technology for Active Intelligent Network Implementation”. *IEEE Network*, 12(3):53–60, 1998.
34. W.H. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, New York, NY, USA, 1998.
35. B. Bruins. “Some Experiences with Emerging Management Technologies”. *The Simple Times*, 4(3):6–8, 1996.
36. M. Brunner and R. Stadler. “Service Management in Multiparty Active Networks”. *IEEE Communications Magazine*, 38(3):144–151, 2000.
37. W. Bumpus. “DMTF Expands Its Role in Developing Information Model Standards”. *Journal of Network and Systems Management*, 6(3):357–360, 1998.
38. W. Bumpus, J.W. Sweitzer, P. Thompson, A.R. Westerinen, and R.C. Williams. *Common Information Model : Implementing the Object Model for Enterprise Management*. Wiley, New York, NY, USA, 2000.
39. R. Burns and M. Quinn. “The Cyber-Agent Framework”. *The Simple Times*, 4(3):12–15, 1996.
40. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley, Chichester, UK, 1996.
41. B. Callaghan, B. Pawlowski, and P. Staubach. *RFC 1813. NFS Version 3 Protocol Specification*. IETF, June 1995.
42. J. Case, M. Fedor, M. Schoffstall, and J. Davin (Eds.). *RFC 1157. A Simple Network Management Protocol (SNMP)*. IETF, May 1990.
43. J. Case, K. McCloghrie, M. Rose, and S. Waldbusser (Eds.). *RFC 1451. Manager-to-Manager Management Information Base*. IETF, April 1993.
44. J. Case, K. McCloghrie, M. Rose, and S. Waldbusser (Eds.). *RFC 1902. Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)*. IETF, January 1996.
45. J. Case, K. McCloghrie, M. Rose, and S. Waldbusser (Eds.). *RFC 1905. Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*. IETF, January 1996.
46. J. Case, K. McCloghrie, M. Rose, and S. Waldbusser (Eds.). *RFC 1906. Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)*. IETF, January 1996.
47. J. Case, K. McCloghrie, M. Rose, and S. Waldbusser (Eds.). *RFC 1907. Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)*. IETF, January 1996.
48. CCITT (now ITU-T). *Recommendation M.3400. TMN management functions.* ITU, Geneva, Switzerland, October 1992.
49. CCITT (now ITU-T). *Recommendation X.700. Data Communication Networks—Management Framework for Open Systems Interconnection (OSI) for CCITT Applications*. ITU, Geneva, Switzerland, September 1992.
50. CCITT (now ITU-T). *Recommendation X.701. Data Communication Networks—Information Technology—Open Systems Interconnection—Systems Management Overview*. ITU, Geneva, Switzerland, January 1992.
51. CCITT (now ITU-T). *Recommendation X.710. Data Communication Networks: Open Systems Interconnection (OSI); Management. Common Management Information Service Definition for CCITT Applications*. ITU, Geneva, Switzerland, March 1991.
52. CCITT (now ITU-T). *Recommendation X.711. Data Communication Networks—Open Systems Interconnection (OSI); Management. Common Management Information Protocol Specification for CCITT Applications*. ITU, Geneva, Switzerland, March 1991.
53. CCITT (now ITU-T). *Recommendation X.722. Data Communication Networks—Information Technology—Open Systems Interconnection—Structure of Information Management: Guidelines for the Definition of Managed Objects*. ITU, Geneva, Switzerland, January 1992.
54. D.B. Chapman and E.D. Zwicky. *Building Internet Firewalls*. O’Reilly & Associates, Sebastopol, CA, USA, 1995.

55. W.R. Cheswick and S.M. Bellovin. *Firewalls and Internet Security—Repelling the Wily Hacker*. Addison-Wesley, Reading, MA, USA, 1994.
56. M.J. Choi, H.T. Ju, H.J. Cha, S.H. Kim, and J.W.K. Hong, "An Efficient and Lightweight Embedded Web Server for Web-based Network Element Management". In *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS 2000), Hawaii, USA, April 2000*, pp. 187–200. IEEE Press, New York, NY, USA, 2000.
57. Cisco. *Release Notes for Cisco IOS Release 12.1*. March 20, 2000. Available at <http://www.cisco.com/univercd/cc/td/doc/product/software/ios121/121relnt/xprn121/121reqs.htm>.
58. J.D. Davidson and S. Ahmed. *Java Servlet API Specification. Version 2.1a*. Sun Microsystems, November 1998.
59. L. Deri. "Surfin' Network Resources Across the Web". In *Proc. IEEE 2nd International Workshop on Systems Management, Toronto, ON, Canada, June 1996*, pp. 158–167. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
60. L. Deri. *HTTP-based SNMP and CMIP Network Management*. Internet-Draft <draft-deri-http-mgmt-00.txt> (now expired). IETF, November 1996.
61. L. Deri. "JLocator: A Web-Based Asset Location System". In A.S. Sethi (Ed.), *Proc. 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'98), Newark, DE, USA, October 1998*, pp. 3–13.
62. P. Deutsch (Ed.). *RFC 1951. DEFLATE Compressed Data Format Specification version 1.3*. IETF, May 1996.
63. DMTF. *Desktop Management Interface Specification*. Version 2.00. March 1996.
64. DMTF. *XML As a Representation for Management Information - A White Paper*. Version 1.0. September 1998. Available at: <http://www.dmtf.org/spec/xmlw.html>.
65. DMTF. *Common Information Model (CIM) Specification*. Version 2.2. June 1999. Available at: <http://www.dmtf.org/spec/cim_spec_v22/>.
66. DMTF. *CIM DTD*. July 1999. Available at: <http://www.dmtf.org/download/spec/xmls/CIM_DTD_V20.txt>.
67. DMTF. *Specification for the Representation of CIM in XML*. Version 2.0. July 1999. Available at: <http://www.dmtf.org/download/spec/xmls/CIM_XML_Mapping20.htm>.
68. DMTF. *Specification for CIM Operations over HTTP*. Version 1.0. August 1999. Available at: <http://www.dmtf.org/download/spec/xmls/CIM_HTTP_Mapping10.htm>.
69. DMTF. *WBEM Initiative*. Available at <http://www.dmtf.org/wbem/>. February 2000.
70. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, MA, USA, 1999.
71. D. Evans. *Supervisory Management: Principles and Practice*. 2nd edition. Cassell Educational Ltd, London, UK, 1986.
72. T. Faber. "ACC: Using Active Networking to Enhance Feedback Congestion Control Mechanisms". *IEEE Network*, 12(3):61–65, 1998.
73. A. Falk and V. Paxson. "Requirements for Unicast Transport/Sessions (RUTS) BOF". In *Proc. 43rd IETF Meeting*, Orlando, FL, USA, December 1998. Available at <http://www.ietf.org/proceedings/98dec/43rd-ietf-98dec-142.html>.
74. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee (Eds.). *RFC 2616. Hypertext Transfer Protocol -- HTTP/1.1*. IETF, June 1999.
75. T. Finin, R. Fritzson, D. McKay, and R. McEntire. "KQML as an Agent Communication Language". In N.R. Adam, B.K. Bhargava, and Y. Yesha (Eds.), *Proc. 3rd International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, MD, USA, November 1994*, pp. 456–463. ACM Press, New York, NY, USA, 1994.
76. FIPA. *Home Page*. Available at <http://www.fipa.org/>. August 2000.
77. FIPA. *FIPA ACL Message Structure Specification*. Revision XC00061D. August 2000.
78. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Menlo Park, CA, USA, 1997.
79. S. Franklin and A. Graesser. "Is it an agent, or just a program?: a taxonomy for autonomous agents". In J.P. Müller, M.J. Wooldridge, and N.R. Jennings (Eds.), *Intelligent Agents III. Proc. ECAI'96 Workshop (ATAL), Budapest, Hungary, August 1996*. LNAI 1193:21–35, Springer, Berlin, Germany, 1997.
80. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. *RFC 2617. HTTP Authentication: Basic and Digest Access Authentication*. IETF, June 1999.
81. N. Freed and N. Borenstein (Eds.). *RFC 2046. Multipurpose Internet Mail Extensions (MIME). Part Two: Media Types*. IETF, November 1996.
82. A. Fuggetta, G.P. Picco, and G. Vigna. "Understanding Code Mobility". *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.
83. J. Galvin and K. McCloghrie (Eds.). *RFC 1445. Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2)*. IETF, April 1993.
84. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Menlo Park, CA, USA, 1994.
85. R.H. Glitho. "Contrasting OSI Systems Management to SNMP and TMN". *Journal of Network and Systems Management*, 6(2):113–133, 1998.
86. GNU. *GNU General Public Licence*. Version 2, June 1991. Available at <http://www.gnu.org/copyleft/gpl.html>.
87. C.F. Goldfarb and P. Prescod. *The XML Handbook*. Prentice Hall, Upper Saddle River, NJ, USA, 1998.
88. G. Goldszmidt. *Distributed Management by Delegation*. Ph.D. thesis, Columbia University, New York, NY, USA, December 1995.

89. G. Goldszmidt and A. Stanford-Clark. “Load Distribution for Scalable Web Servers: Summer Olympics 1996—A Case Study”. In A. Seneviratne, V. Varadarajan, and P. Ray (Eds.), *Proc. 8th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM’97), Sydney, Australia, October 1997*, pp. 53–64.
90. J. Gosling and H. McGilton. *The Java Language Environment: a White Paper*. Sun Microsystems, October 1995.
91. L.E. Greiner. “Evolution and Revolution as Organizations Grow”. *Harvard Business Review*, 50(4):37–46, 1972.
92. C. Harler. *Web-Based Network Management: Beyond the Browser*. Wiley, New York, NY, USA, 1999.
93. D. Harrington, R. Presuhn, and B. Wijnen (Eds.). *RFC 2571. An Architecture for Describing SNMP Management Frameworks*. IETF, May 1999.
94. B. Harrison, P.E. Mellquist, and A. Pell. *Web Based System and Network Management*. Internet-Draft <draft-mellquist-web-sys-01.txt> (now expired). IETF, November 1996.
95. M.J. Hatch. *Organization Theory: Modern, Symbolic, and Postmodern Perspectives*. Oxford University Press, Oxford, UK, 1997.
96. M. Hauswirth and M. Jazayeri. “A Component and Communication Model for Push Systems”. In O. Nierstrasz and M. Lemoine (Eds.), *Software Engineering. Proc. 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’99)*, Toulouse, France, September 1999. LNCS 1687:20–38, Springer, Berlin, Germany, 1999.
97. H.G. Hegering and S. Abeck. *Integrated Network and System Management*. Addison-Wesley, Wokingham, UK, 1994.
98. H.G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems: Concepts, Architectures, and Their Operational Application*. Morgan Kaufmann, San Francisco, CA, USA, 1999.
99. J.R. Hess, D.C. Lee, S.J. Harper, M.T. Jones, and P.M. Athanas. “Implementation and Evaluation of a Prototype Reconfigurable Router”. In *Proc. 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’99), Napa, CA, USA, April 1999*, pp. 44–50.
100. J.W.K. Hong, J.Y. Kong, T.H. Yun, J.S. Kim, J.T. Park, and J.W. Baek. “Web-Based Intranet Services and Network Management”. *IEEE Communications Magazine*, 35(10):100–110, 1997.
101. J.W.K. Hong, S.S. Kwon, and J.Y. Kim. WebTrafMon: Web-based Internet/Intranet network traffic monitoring and analysis system. *Computer Communications*, 22(14):1333–1342, 1999.
102. HTTP-WG mailing list. Thread entitled “*Drawbacks of persistent connections*”. Available at: <http://www.ics.uci.edu/pub/ietf/http/hypermail/1998q2/0191.html>. June 1998.
103. IANA. *Protocol Numbers and Assignment Services*. Available at <http://www.iana.org/numbers.html>. This Web site updates RFC 1700 which is now obsolete.
104. IBM. *AlphaWorks*. March 1999. Available at <http://www.alphaworks.ibm.com/>.
105. ISO/IEC JTC1/SC21 N6131. *ISO/IEC Working Draft 8824-3. Information Technology—ASN.1 Encoding Rules—LightWeight Encoding Rules (LWER)*. ISO, Geneva, Switzerland, 1994. *(Cited in [147].)*
106. ISO WG4 N1851. *Open Distributed Management Architecture*. Working Draft 3. ISO, July 1995.
107. V. Issarny, L. Bellissard, M. Riveill, and A. Zarras. “Component-Based Programming of Distributed Applications”. In S. Krakowiak and S. Shrivastava (Eds.), *Advances in Distributed Systems—Advanced Distributed Computing: From Algorithms to Systems*. LNCS 1752:327–353, Springer, Berlin, Germany, 2000.
108. ITU-T. *Recommendation M.3010. Principles for a Telecommunications management network.* ITU, Geneva, Switzerland, May 1996.
109. ITU-T. *Recommendation X.500. Information technology—Open Systems Interconnection—The Directory: Overview of concepts, models and services*. ITU, Geneva, Switzerland, August 1997.
110. 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, Geneva, Switzerland, July 1994.
111. ITU-T. *Recommendation X.691. Information Technology—ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER)*. ITU, Geneva, Switzerland, April 1995.
112. ITU-T. *Recommendation X.725. Data Networks and Open System Communications—OSI Management—Information Technology—Open Systems Interconnection—Structure of Information Management: General Relationship Model*. ITU, Geneva, Switzerland, November 1995.
113. ITU-T. *Recommendation X.753. Information technology—Open Systems Interconnection—Systems management: Command sequencer for systems management*. ITU, Geneva, Switzerland, October 1997.
114. A. John, K. Vanderveen, and B. Sugla. "An XML-Based Framework for Dynamic SNMP MIB Extension". In R. Stadler and B. Stiller (Eds.), *Active Technologies for Network and Service Management. Proc. 10th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM’99), Zurich, Switzerland, October 1999*. LNCS 1700:107–120, Springer, Berlin, Germany, 1999.
115. H.T. Ju, M.J. Choi, S.H. Kim, H.J. Cha, and J.W.K. Hong. "Embedded Web Server Technology for Web-based Network Element Management". In *Proc. Asia-Pacific Network Operations and Management Symposium (APNOMS’99), Kyongju, Korea, September 1999*, pp. 317–332.
116. G. Karjoth, N. Asokan, and C. Gülcü. “Protecting the Computation Results of Free-Roaming Agents”. In K. Rothermel and F. Hohl (Eds.). *Proc. 2nd International Workshop on Mobile Agents (MA’98), Stuttgart, Germany, September 1998*. LNCS 1477:195–207, Springer, Berlin, Germany, 1998.
117. W. Kasteleijn. *Web-Based Management*. M.S. thesis, University of Twente, Enschede, The Netherlands, April 1997.
118. R. Kawamura and R. Stadler. “Active Distributed Management for IP Networks”. *IEEE Communications Magazine*, 38(4):114–120, 2000.

119. D. Kegel. *The C10K Problem*. Available at <http://www.kegel.com/c10k.html>, April 2000.
120. M. Knapik and J. Johnson. *Developing Intelligent Agents for Distributed Systems*. McGraw Hill, New York, NY, USA, 1998.
121. J. Kramer. “Chapter 3. Distributed Systems”. In M. Sloman (Ed.). *Network and Distributed Systems Management*, pp. 47–66. Addison-Wesley, Wokingham, UK, 1994.
122. G.P. Kumar and P. Venkataram. “Artificial intelligence approaches to network management: recent advances and a survey”. *Computer Communications*, 20(15):1313–1322, 1997.
123. C. Larman. *Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.
124. A. Leinwand and K. Fang Conroy. *Network Management: a Practical Perspective*. 2nd edition. Addison-Wesley, Reading, MA, USA, 1996.
125. D. Levi, P. Meyer, and B. Stewart (Eds.). *RFC 2573. SNMPv3 Applications.* IETF, April 1999.
126. D. Levi and J. Schönwälder (Eds.). *RFC 2592. Definitions of Managed Objects for the Delegation of Management Scripts*. IETF, May 1999.
127. D. Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. O’Reilly & Associates, Sebastopol, CA, USA, 1994.
128. J. Lindsay. *The Web Based Management Page*. Available at <http://www.mindspring.com/~jlindsay/webbased.html>. July 2000.
129. Linux-kernel mailing list. Thread entitled “*> 15,000 Simultaneous Connections*”. Available at: <http://kernelnotes.org/lnxlists/linux-kernel/lk_9909_01/msg00783.html>. September 1999.
130. J.P. Martin-Flatin and S. Znaty. “Annotated Typology of Distributed Network Management Paradigms”. Technical Report SSC/1997/008, SSC, EPFL, Lausanne, Switzerland, March 1997.
131. J.P. Martin-Flatin and S. Znaty. “A Simple Typology of Distributed Network Management Paradigms”. In A. Seneviratne, V. Varadarajan, and P. Ray (Eds.), *Proc. 8th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM’97), Sydney, Australia, October 1997*, pp. 13–24.
132. J.P. Martin-Flatin. *IP Network Management Platforms Before the Web*. Technical Report SSC/1998/021, version 2, SSC, EPFL, Lausanne, Switzerland, December 1998.
133. J.P. Martin-Flatin, S. Znaty, and J.P. Hubaux. “A Survey of Distributed Enterprise Network and Systems Management”. *Journal of Network and Systems Management*, 7(1):9–26, 1999.
134. J.P. Martin-Flatin. “Push vs. Pull in Web-Based Network Management”. In M. Sloman, S. Mazumdar, and E. Lupu (Eds.), *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM’99), Boston, MA, USA, May 1999*, pp. 3–18. IEEE, Piscataway, NJ, USA, 1999.
135. J.P. Martin-Flatin, L. Bovet, and J.P. Hubaux. "JAMAP: a Web-Based Management Platform for IP Networks". In R. Stadler and B. Stiller (Eds.), *Active Technologies for Network and Service Management. Proc. 10th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM’99), Zurich, Switzerland, October 1999*. LNCS 1700:164–178, Springer, Berlin, Germany, 1999.
136. J.P. Martin-Flatin. *JAMAP 0.3*. Available at <http://ica2www.epfl.ch/~jpmf/projects/jamap/>. April 2000.
137. M.C. Maston. “Using the World Wide Web and Java for Network Service Management”. In A. Lazar, R. Saracco, and R. Stadler (Eds.), *Integrated Network Management V. Proc. 5th IFIP/IEEE International Symposium on Integrated Network Management (IM’97), San Diego, CA, USA, May 1997*, pp. 71–84. Chapman & Hall, London, UK, 1997.
138. S. Mazumdar. “Inter-Domain Management between CORBA and SNMP”. In *Proc. 7th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM’96), L’Aquila, Italy, October 1996*.
139. S. Mazumdar and T. Roberts (Eds.). *Translation of GDMO Specification into CORBA-IDL*. Report of the XoJIDM task force, August 1995.
140. S. Mazumdar (Ed.). *Translation of SNMPv2 Specification into CORBA-IDL*. Report of the XoJIDM task force, September 1996.
141. K. McCloghrie. “The SNMP Framework”. *The Simple Times*, 4(1):9–10, 1996.
142. K. McCloghrie and J. Galvin (Eds.). *RFC 1447. Party MIB for version 2 of the Simple Network Management Protocol (SNMPv2).* IETF, April 1993.
143. K. McCloghrie and M. Rose (Eds.). *RFC 1213. Management Information Base for Network Management of TCP/IP-based internets: MIB-II*. IETF, March 1991.
144. D. Megginson. *Structuring XML Documents*. Prentice Hall, Upper Saddle River, NJ, USA, 1998.
145. P. Merle, C. Gransart, and J.M. Geib. “CorbaWeb: A Generic Object Navigator”. In *Proc. 5th International World Wide Web Conference (WWW5), Paris, France, May 1996*. Elsevier, Amsterdam, The Netherlands, 1996.
146. K. Meyer, M. Erlinger, J. Betser, C. Sunshine, G. Goldszmidt, and Y. Yemini. “Decentralizing Control and Intelligence in Network Management”. In A.S. Sethi, Y. Raynaud, and F. Faure-Vincent (Eds.), *Integrated Network Management IV. Proc. 4th IFIP/IEEE International Symposium on Integrated Network Management (ISINM’95), Santa Barbara, CA, USA, May 1995*, pp. 4–16. Chapman & Hall, London, UK, 1995.
147. N. Mitra. “Efficient Encoding Rules for ASN.1-Based Protocols”. *AT&T Technical Journal*, 73(3):80–93, 1994.
148. J. Mogul and S. Deering (Eds.). *RFC 1191. Path MTU Discovery*. IETF, November 1990.
149. M.A. Mountzia. “A distributed management approach based on flexible agents”. *Interoperable Communication Networks*, 1:99–120, 1998.
150. P. Mullaney. “Overview of a Web-Based Agent”. *The Simple Times*, 4(3):8–12, 1996.
151. N.J. Muller and L.L. Muller. “Using the World Wide Web for Network Support”. *International Journal of Network Management*, 5(6):326–341, 1995.

152. L.J. Mullins. *Management and Organisational Behaviour*. 2nd edition. Pitman, London, UK, 1989.
153. Netscape. *An Exploration of Dynamic Documents*. 1995. Available at <http://home.mcom.com/assist/net_sites/pushpull.html>.
154. G. Neufeld and S. Vuong. "An overview of ASN.1". *Computer Networks and ISDN Systems*, 23:393–415, 1992.
155. H. Nielsen, P. Leach, and S. Lawrence. *RFC 2774. An HTTP Extension Framework*. IETF, February 2000.
156. T. Oetiker and D. Rand. *MRTG: Multi-Router Traffic Grapher*. Available at <http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/mrtg.html>, July 2000.
157. J.L. Oliveira and J.A. Martins. "A Management Architecture Based on Network Topology Information". Journal of Network andSystems Management, 2(4):401–414, 1994.
158. OMG. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0. July 1995.
159. S. Ortiz. "Active Networks: The Programmable Pipeline". *IEEE Computer*, 31(8):19–21, 1998.
160. J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.
161. G. Pavlou. *Telecommunications Management Network: a Novel Approach Towards its Architecture and Realisation Through Object-Oriented Software Platforms*. Ph.D. Thesis, Dept. of Computer Science, University College London, UK, March 1998.
162. G. Pavlou. "Chapter 2. OSI Systems Management, Internet SNMP, and ODP/OMG CORBA as Technologies for Telecommunications Network Management". In S. Aidarous and T. Plevyak (Eds.), *Telecommunications Network Management: Technologies and Implementations*, pp. 63–110. IEEE Press, New York, NY, USA, 1998.
163. G. Pavlou. "Using Distributed Object Technologies in Telecommunications Network Management". *IEEE Journal on Selected Areas in Communications*, 18(5):644–653, 2000.
164. J. Pavón, J. Tomás, Y. Bardout, and L.H. Hauw. "CORBA for Network and Service Management in the TINA Framework". *IEEE Communications Magazine*, 36(3):72–79, 1998.
165. J. Pavón. "Building Telecommunications Management Applications with CORBA". *IEEE Communications Surveys*, 2(2):2–16, 1999.
166. D.T. Perkins. "SNMP Versions". *The Simple Times*, 5(1):13–14, 1997.
167. D.T. Perkins. "Questions Answered". *The Simple Times*, 6(1):13–17, 1998.
168. M. Post, C.C. Shen, and J. Wei. "The Manager/Agency Paradigm for Distributed Network Management". In *Proc. 1996 IEEE Network Operations and Management Symposium (NOMS'96), Kyoto, Japan, April 1996*, 1:44–53. IEEE, Piscataway, NJ, USA, 1996.
169. J. Postel. *RFC 768. User Datagram Protocol*. IETF, August 1980.
170. K. Psounis. "Active Networks: Applications, Security, Safety, and Architectures ". *IEEE Communications Surveys*, 2(1):2–16, 1999.
171. A.S. Rao and M.P. Georgeff. "Modeling rational agents within a BDI-architecture". In R. Fikes and E. Sandewall (Eds.), *Proc. Knowledge Representation and Reasoning (KR&R-91), San Mateo, CA, USA, April 1991*, pp. 473–484. Morgan Kaufmann, 1991.
172. D. Raz and Y. Shavitt. "Active Networks for Efficient Distributed Network Management". *IEEE Communications Magazine*, 38(3):138–143, 2000.
173. A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, Reading, MA, USA, 1996.
174. A.I. Rivière. *GEMINI: A Generic Environment for Management Information Integration*. Ph.D. thesis, P. Sabatier University of Toulouse, France, December 1997.
175. A.I. Rivière and M. Sibilla. "Management Information Models Integration: From Existing Approaches to New Unifying Guidelines". *Journal of Network and Systems Management*, 6(3):333–356, 1998.
176. M.T. Rose. *The Simple Book: an Introduction to Networking Management*. Revised 2nd edition. Prentice Hall, Upper Saddle River, NJ, USA, 1996.
177. M.T. Rose and K. McCloghrie (Eds.). *RFC 1155. Structure and Identification of Management Information for TCP/IP-based Internets*. IETF, May 1990.
178. M. Rose and K. McCloghrie (Eds.). *RFC 1212. Concise MIB Definitions*. IETF, March 1991.
179. T. Sander and C Tschudin. "Protecting Mobile Agents Against Malicious Hosts". In G. Vigna (Ed.). *Mobile Agents and Security*. LNCS 1419:44–60, Springer, Berlin, Germany, 1998.
180. J. Saperia and J. Schönwälder (Eds.). *Policy-Based Enhancements to the SNMP Framework*. Internet-Draft <draft-schoenw-policy-snmp-00.txt> (now expired). IETF, September 1999.
181. J. Schönwälder. "Network management by delegation—From research prototypes towards standards". *Computer Networks and ISDN Systems*, 29(15):1843–1852, 1997.
182. J. Schönwälder. "Emerging Internet Management Standards". Tutorial given at the *6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*, Boston, MA, USA, May 1999. Available at: <http://www.ibr.cs.tu-bs.de/~schoenw/slides/im-99.ps.gz>.
183. J. Schönwälder and F. Strauss. "Next Generation Structure of Management Information for the Internet". In R. Stadler and B. Stiller (Eds.), *Active Technologies for Network and Service Management. Proc. 10th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'99), Zurich, Switzerland, October 1999*. LNCS 1700:93–106, Springer, Berlin, Germany, 1999.
184. J. Schönwälder (Ed.). *SNMP over TCP Transport Mapping*. Internet-Draft <draft-irtf-nmrg-snmp-tcp-04.txt> (work in progress). IETF, April 2000.
185. J. Schönwälder, J. Quittek, and C. Kappler. "Building Distributed Management Applications with the IETF Script MIB". *IEEE Journal on Selected Areas in Communications*, 18(5):702–714, 2000.

186. J. Schönwälder and F. Strauss. *Using XML to Exchange SMI Definitions*. Internet-Draft <draft-irtf-nmrg-smi-xml-00.txt> (work in progress). IETF, June 2000.
187. J. Schönwälder. *Scotty—Tcl Extensions for Network Management Applications*. Available at <http://wwwhome.cs.utwente.nl/~schoenw/scotty/>. July 2000.
188. B. Schwartz, A.W. Jackson, W.T. Strayer, W. Zhou, R.D. Rockwell, and C. Partridge. “Smart Packets: Applying Active Networks to Network Management”. *ACM Transactions on Computer Systems*, 18(1):67–88, 2000.
189. J. Semke, J. Mahdavi, and M. Mathis. “Automatic TCP Buffer Tuning”. In *Proc. ACM SIGCOMM’98, Vancouver, BC, Canada, September 1998. ACM Computer Communication Review*, 28(4):315–323, 1998.
190. J.K. Shrewsbury. “An Introduction to TMN”. *Journal of Network and Systems Management*, 3(1):13–38, 1995.
191. J. Siegel. *CORBA Fundamentals and Programming*. Wiley, New York, NY, USA, 1996.
192. SimpleWeb. *Commercial SNMP / Network Management Software*. Available at <http://wwwsnmp.cs.utwente.nl/software/commercial.html>. July 2000.
193. SimpleWeb. *The IETF MIBs*. Available at <http://www.simpleweb.org/ietf/mibs/>. July 2000.
194. M. Sloman. “Policy Driven Management for Distributed Systems”. *Journal of Network and Systems Management*, 2(4):333–360, 1994.
195. M. Sloman (Ed.). *Network and Distributed Systems Management*. Addison-Wesley, Wokingham, UK, 1994.
196. M. Sloman and K. Twidle. “Chapter 16. Domains: A Framework for Structuring Management Policy”. In M. Sloman (Ed.). *Network and Distributed Systems Management*, pp. 433–453. Addison-Wesley, Wokingham, UK, 1994.
197. R. Smith. *Internet Cryptography*. Addison-Wesley, Reading, MA, USA, 1997.
198. J.M. Smith, K.L. Calvert, S.L. Murphy, H.K. Orman, and L.L. Peterson. “Activating Networks: A Progress Report”. *IEEE Computer*, 32(4):32–41, 1999.
199. F. Somers. “HYBRID: Unifying Centralised and Distributed Network Management using Intelligent Agents”. In *Proc. 1996 IEEE Network Operations and Management Symposium (NOMS’96), Kyoto, Japan, April 1996*. 1:34–43. IEEE, Piscataway, NJ, USA, 1996.
200. S.E. Spero. *Analysis of HTTP Performance Problems*. June 1995. Available at <http://www.w3.org/Protocols/HTTP-NG/http-prob.html>.
201. R. Sprenkels and J.P. Martin-Flatin. "Bulk Transfers of MIB Data". *The Simple Times*, 7(1):1–7, March 1999.
202. P. Sridharan. *Advanced Java networking*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
203. M. St. Pierre, J. Fullton, K. Gamiel, J. Goldman, B. Kahle, J. Kunze, H. Morris, and F. Schiettecatte (Eds.). *RFC 1625. WAIS over Z39.50-1988*. IETF, June 1994.
204. W. Stallings. *SNMP, SNMPv2, and CMIP: the Practical Guide to Network Management Standards*. Addison-Wesley, Reading, MA, USA, 1993. *(Obsoleted by [207], except for CMIP)*.
205. W. Stallings. “SSL: Foundation for Web Security”. *The Internet Protocol Journal*, 1(1):20–29, 1998.
206. W. Stallings. “SNMPv3: A Security Enhancement to SNMP”. *IEEE Communications Surveys*, 1(1):2–17, 1998.
207. W. Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Third edition. Addison-Wesley, Reading, MA, USA, 1999.
208. W. Stallings. *Network Security Essentials: Applications and Standards*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.
209. W. Stallings. “IP Security”. *The Internet Protocol Journal*, 3(1):11–26, 2000.
210. J.W. Stamos and D.K. Gifford. “Remote Evaluation”. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, 1990.
211. W.R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, MA, USA, 1994.
212. W.R. Stevens. *Unix Network Programming, Volume 1*. 2nd edition. Prentice Hall, Upper Saddle River, NJ, USA, 1998.
213. R.R. Stewart, Q. Xie, K. Morneault, C. Sharp, H.J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson (Eds.). *Stream Control Transmission Protocol*. Internet-Draft <draft-irtf-sigtran-sctp-08.txt> (work in progress). IETF, April 2000.
214. J. Strassner. *Directory Enabled Networks*. Macmillan, Indianapolis, IN, USA, 1999.
215. Sun Microsystems. *RFC 1094. NFS: Network File System Protocol Specification*. IETF, March 1989.
216. Sun Microsystems. *Jini Technology Executive Overview*. Revision 1.0. January 1999.
217. Sun Microsystems. *Java Development Kit 1.1*. March 1999. Available at <http://www.javasoft.com/products/jdk/1.1/>.
218. Sun Microsystems. *Java Management Extensions White Paper*. Revision 01. June 1999.
219. Sun Microsystems. *Java Management Extensions Instrumentation and Agent Specification, v1.0*. December 1999.
220. Sun Microsystems. *Federated Management Architecture (FMA) Specification Version 1.0*. Revision 0.4. January 2000
221. Sunsoft. *Java Management API Architecture*. Revision A. September 1996.
222. C. Szyperski. *Component Software : Beyond Object-Oriented Programming.* Addison-Wesley, Harlow, UK, 1998.
223. D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. “A Survey of Active Network Research”. *IEEE Communications Magazine*, 35(1):80–86, 1997.
224. D.L. Tennenhouse and D. Wetherall. “Towards an Active Network Architecture”. *ACM Computer Communication Review*, 26(2):5–18, 1996.
225. K. Terplan. *Web-Based Systems & Network Management*. CRC Press, Boca Raton, Florida, USA, 1999.
226. J.P. Thompson. “Web-Based Enterprise Management Architecture”. *IEEE Communications Magazine*, 36(3):80–86, 1998.
227. R. Tschalär. *HTTPClient*. March 1999. Available at <http://www.innovation.ch/java/HTTPClient/>.
228. UCB. *UCB/LBNL/VINT Network Simulator - ns (version 2)*. Available at <http://www-mash.cs.berkeley.edu/ns/>. May 2000.
229. G. Vigna (Ed.). *Mobile Agents and Security*. LNCS 1419, Springer, Berlin, Germany, 1998.

230. W3C. *Extensible Markup Language (XML) 1.0. W3C Recommendation REC-xml-19980210*. Available at <http://http://www.w3.org/TR/1998/REC-xml-19980210>. February 1998.
231. W3C. *Namespaces in XML. W3C Recommendation REC-xml-names-19990114*. Available at <http://www.w3.org/TR/1999/REC-xml-names-19990114/>. January 1999.
232. W3C. *Document Object Model (DOM)*. Available at <http://www.w3.org/DOM/>. July 2000.
233. W3C. *Extensible Markup Language (XML)*. Available at <http://www.w3.org/XML/>. July 2000.
234. S. Waldbusser (Ed.). *RFC 1271. Remote Network Monitoring Management Information Base*. IETF, November 1991.
235. S. Waldbusser (Ed.). *RFC 1757. Remote Network Monitoring Management Information Base*. IETF, February 1995.
236. S. Waldbusser (Ed.). *RFC 2021. Remote Network Monitoring Management Information Base Version 2 using SMIv2*. IETF, January 1997.
237. S. Waldbusser and P. Grillo (Eds.). *RFC 2790. Host Resources MIB*. IETF, March 2000.
238. L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl*. 2nd edition. O'Reilly & Associates, Sebastopol, CA, USA, 1996.
239. G. Waters (Ed.). *RFC 1910. User-based Security Model for SNMPv2*. IETF, February 1996.
240. Web Developer's Virtual Library. *XML Specifications, Proposals and Vocabularies*. Available at <http://wdvl.com/Authoring/Languages/XML/Specifications.html>. July 2000.
241. T.D. Weinshall and Y.A. Raveh. *Managing Growing Organizations: A New Approach*. Wiley, Chichester, UK, 1983.
242. C. Wellens and K. Auerbach. "Towards Useful Management". *The Simple Times*, 4(3):1–6, 1996.
243. M. Welsh, S.D. Gribble, E.A. Brewer, and D. Culler. *A Design Framework for Highly Concurrent Systems*. Technical Report UCB/CSD-00-1108, Computer Science Division, U.C. Berkeley, April 2000.
244. R. Wies. "Policies in Network and Systems Management—Formal Definition and Architecture". *Journal of Network and Systems Management*, 2(1):63–83, 1994.
245. I. Wijegunaratne and G. Fernandez. *Distributed Applications Engineering: Building New Applications and Managing Legacy Applications with Distributed Technologies*. Springer, London, UK, 1998.
246. B. Wijnen, R. Presuhn, and K. McCloghrie (Eds.). *RFC 2575. View-Based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)*. IETF, April 1999.
247. E. Wilde. *Wilde's WWW. Technical Foundations of the World Wide Web*. Springer, Berlin, Germany, 1999.
248. M. Wooldridge and N.R. Jennings. "Agent Theories, Architectures and Languages: a Survey". In M. Wooldridge and N.R. Jennings (Eds.). *Intelligent Agents. Proc. ECAI-94, Workshop on Agent Theories, Architectures and Languages, Amsterdam, The Netherlands, August 1994*. LNAI 890:1–39. Springer, Berlin, Germany, 1995.
249. G.R. Wright and W.R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley, Reading, MA, USA, 1995.
250. Y. Yemini. "The OSI Network Management Model". *IEEE Communications Magazine*, 31(5):20–29, 1993.
251. Y. Yemini, G. Goldszmidt, and S. Yemini. "Network Management by Delegation". In I. Krishnan and W. Zimmer (Eds.), *Proc. IFIP 2nd International Symposium on Integrated Network Management (ISINM'91), Washington, DC, USA, April 1991*, pp. 95–107. North-Holland, Elsevier, Amsterdam, The Netherlands, 1991.
252. T. Yemini and S. da Silva. "Towards Programmable Networks". In *Proc. 7th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'96), L'Aquila, Italy, October 1996*.
253. T. Zhang, S. Covaci, and R. Popescu-Zeletin. "Intelligent Agents in Network and Service Management". In *Proc. IEEE Global Telecommunications Conference (GLOBECOM'96), London, UK, November 1996*. 3:1855–1861. IEEE, Piscataway, NJ, USA, 1996.
254. S. Znaty and O. Cherkaoui. "IDEAL: An Integrated Resource Management Language". In *Proc. IEEE Global Telecommunications Conference (GLOBECOM'97), Phoenix, AZ, USA, November 1997*. 1:202–206. IEEE, Piscataway, NJ, USA, 1997.

Appendix A

THE INTERFACES GROUP IN SNMP MIB-II

In this appendix, we include the definition of the Interfaces Group in SNMP MIB-II [143, pp. 16–23]. This is used in Chapter 8.

```
-- the Interfaces group

-- Implementation of the Interfaces group is mandatory for
-- all systems.

ifNumber OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The number of network interfaces (regardless of
            their current state) present on this system."
    ::= { interfaces 1 }


-- the Interfaces table

-- The Interfaces table contains information on the entity's
-- interfaces.  Each interface is thought of as being
-- attached to a 'subnetwork'.  Note that this term should
-- not be confused with 'subnet' which refers to an
-- addressing partitioning scheme used in the Internet suite
-- of protocols.

ifTable OBJECT-TYPE
```

```
    SYNTAX  SEQUENCE OF IfEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
            "A list of interface entries.  The number of
            entries is given by the value of ifNumber."
    ::= { interfaces 2 }

ifEntry OBJECT-TYPE
    SYNTAX  IfEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
            "An interface entry containing objects at the
            subnetwork layer and below for a particular
            interface."
    INDEX   { ifIndex }
    ::= { ifTable 1 }

IfEntry ::=
    SEQUENCE {
        ifIndex
            INTEGER,
        ifDescr
            DisplayString,
        ifType
            INTEGER,
        ifMtu
            INTEGER,
        ifSpeed
            Gauge,
        ifPhysAddress
            PhysAddress,
        ifAdminStatus
            INTEGER,
        ifOperStatus
            INTEGER,
        ifLastChange
            TimeTicks,
        ifInOctets
            Counter,
        ifInUcastPkts
            Counter,
        ifInNUcastPkts
            Counter,
        ifInDiscards
            Counter,
        ifInErrors
            Counter,
        ifInUnknownProtos
            Counter,
        ifOutOctets
```

```
            Counter,
        ifOutUcastPkts
            Counter,
        ifOutNUcastPkts
            Counter,
        ifOutDiscards
            Counter,
        ifOutErrors
            Counter,
        ifOutQLen
            Gauge,
        ifSpecific
            OBJECT IDENTIFIER
    }

ifIndex OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory

    DESCRIPTION
            "A unique value for each interface.  Its value
            ranges between 1 and the value of ifNumber.  The
            value for each interface must remain constant at
            least from one re-initialization of the entity's
            network management system to the next re-
            initialization."
    ::= { ifEntry 1 }

ifDescr OBJECT-TYPE
    SYNTAX  DisplayString (SIZE (0..255))
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "A textual string containing information about the
            interface.  This string should include the name of
            the manufacturer, the product name and the version
            of the hardware interface."
    ::= { ifEntry 2 }

ifType OBJECT-TYPE
    SYNTAX  INTEGER {
                other(1),          -- none of the following
                regular1822(2),
                hdh1822(3),
                ddn-x25(4),
                rfc877-x25(5),
                ethernet-csmacd(6),
                iso88023-csmacd(7),
                iso88024-tokenBus(8),
                iso88025-tokenRing(9),
                iso88026-man(10),
```

```
                    starLan(11),
                    proteon-10Mbit(12),
                    proteon-80Mbit(13),
                    hyperchannel(14),
                    fddi(15),
                    lapb(16),
                    sdlc(17),
                    ds1(18),            -- T-1
                    e1(19),             -- european equiv. of T-1
                    basicISDN(20),
                    primaryISDN(21),    -- proprietary serial
                    propPointToPointSerial(22),
                    ppp(23),
                    softwareLoopback(24),
                    eon(25),             -- CLNP over IP [11]
                    ethernet-3Mbit(26),
                    nsip(27),            -- XNS over IP
                    slip(28),            -- generic SLIP
                    ultra(29),           -- ULTRA technologies
                    ds3(30),             -- T-3
                    sip(31),             -- SMDS
                    frame-relay(32)
                }
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The type of interface, distinguished according to
            the physical/link protocol(s) immediately 'below'
            the network layer in the protocol stack."
    ::= { ifEntry 3 }

ifMtu OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The size of the largest datagram which can be
            sent/received on the interface, specified in
            octets.  For interfaces that are used for
            transmitting network datagrams, this is the size
            of the largest network datagram that can be sent
            on the interface."
    ::= { ifEntry 4 }

ifSpeed OBJECT-TYPE
    SYNTAX  Gauge
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "An estimate of the interface's current bandwidth
            in bits per second.  For interfaces which do not
            vary in bandwidth or for those where no accurate
```

```
                estimation can be made, this object should contain
                the nominal bandwidth."
        ::= { ifEntry 5 }

    ifPhysAddress OBJECT-TYPE
        SYNTAX  PhysAddress
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The interface's address at the protocol layer
                immediately 'below' the network layer in the
                protocol stack.  For interfaces which do not have
                such an address (e.g., a serial line), this object
                should contain an octet string of zero length."
        ::= { ifEntry 6 }
    ifAdminStatus OBJECT-TYPE
        SYNTAX  INTEGER {
                    up(1),       -- ready to pass packets
                    down(2),
                    testing(3)   -- in some test mode
                }
        ACCESS  read-write
        STATUS  mandatory
        DESCRIPTION
                "The desired state of the interface.  The
                testing(3) state indicates that no operational
                packets can be passed."
        ::= { ifEntry 7 }

    ifOperStatus OBJECT-TYPE
        SYNTAX  INTEGER {
                    up(1),       -- ready to pass packets
                    down(2),
                    testing(3)   -- in some test mode
                }
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The current operational state of the interface.
                The testing(3) state indicates that no operational
                packets can be passed."
        ::= { ifEntry 8 }

    ifLastChange OBJECT-TYPE
        SYNTAX  TimeTicks
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The value of sysUpTime at the time the interface
                entered its current operational state.  If the
                current state was entered prior to the last re-
                initialization of the local network management
```

```
                subsystem, then this object contains a zero
                value."
        ::= { ifEntry 9 }

    ifInOctets OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The total number of octets received on the
                interface, including framing characters."
        ::= { ifEntry 10 }

    ifInUcastPkts OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The number of subnetwork-unicast packets
                delivered to a higher-layer protocol."
        ::= { ifEntry 11 }

    ifInNUcastPkts OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The number of non-unicast (i.e., subnetwork-
                broadcast or subnetwork-multicast) packets
                delivered to a higher-layer protocol."
        ::= { ifEntry 12 }

    ifInDiscards OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The number of inbound packets which were chosen
                to be discarded even though no errors had been
                detected to prevent their being deliverable to a
                higher-layer protocol.  One possible reason for
                discarding such a packet could be to free up
                buffer space."
        ::= { ifEntry 13 }

    ifInErrors OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The number of inbound packets that contained
                errors preventing them from being deliverable to a
```

```
                higher-layer protocol."
        ::= { ifEntry 14 }

    ifInUnknownProtos OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The number of packets received via the interface
                which were discarded because of an unknown or
                unsupported protocol."
        ::= { ifEntry 15 }

    ifOutOctets OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The total number of octets transmitted out of the
                interface, including framing characters."
        ::= { ifEntry 16 }

    ifOutUcastPkts OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The total number of packets that higher-level
                protocols requested be transmitted to a
                subnetwork-unicast address, including those that
                were discarded or not sent."
        ::= { ifEntry 17 }

    ifOutNUcastPkts OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The total number of packets that higher-level
                protocols requested be transmitted to a non-
                unicast (i.e., a subnetwork-broadcast or
                subnetwork-multicast) address, including those
                that were discarded or not sent."
        ::= { ifEntry 18 }

    ifOutDiscards OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The number of outbound packets which were chosen
                to be discarded even though no errors had been
```

```
                detected to prevent their being transmitted.  One
                possible reason for discarding such a packet could
                be to free up buffer space."
        ::= { ifEntry 19 }

    ifOutErrors OBJECT-TYPE
        SYNTAX  Counter
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The number of outbound packets that could not be
                transmitted because of errors."
        ::= { ifEntry 20 }
    ifOutQLen OBJECT-TYPE
        SYNTAX  Gauge
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "The length of the output packet queue (in
                packets)."
        ::= { ifEntry 21 }

    ifSpecific OBJECT-TYPE
        SYNTAX  OBJECT IDENTIFIER
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
                "A reference to MIB definitions specific to the
                particular media being used to realize the
                interface.  For example, if the interface is
                realized by an ethernet, then the value of this
                object refers to a document defining objects
                specific to ethernet.  If this information is not
                present, its value should be set to the OBJECT
                IDENTIFIER { 0 0 }, which is a syntatically valid
                object identifier, and any conformant
                implementation of ASN.1 and BER must be able to
                generate and recognize this value."
        ::= { ifEntry 22 }
```

Appendix B

METAMODEL-LEVEL XML MAPPING OF THE INTERFACES GROUP IN SNMP MIB-II

In this appendix, we include the metamodel-level XML mapping of the Interfaces Group in SNMP MIB-II. This document is an excerpt of the representation of MIB-II module definitions in XML. It was generated by using the SimpleWeb IETF MIB converter [193] maintained by the University of Twente, The Netherlands and the Technical University of Braunschweig, Germany. This mapping is used in Chapter 8.

```
<?xml version=”1.0”?>
<!DOCTYPE smi:smi SYSTEM “/ietf/mibs/modules/xml/smi.dtd”>

<!-- This module has been generated by smidump 0.2.4.
     then manually edited by J.P. Martin-Flatin -->

<smi xmlns:smi=”http://www.irtf.org/nmrg/”>
  <module name=”RFC1213-MIB” language=”SMIv1”>
  </module>

  <imports>
    <import module=”RFC1155-SMI” name=”mgmt”/>
    <import module=”RFC1155-SMI” name=”NetworkAddress”/>
    <import module=”RFC1155-SMI” name=”IpAddress”/>
    <import module=”RFC1155-SMI” name=”Counter”/>
    <import module=”RFC1155-SMI” name=”Gauge”/>
    <import module=”RFC1155-SMI” name=”TimeTicks”/>
    <import module=”RFC1212-MIB” name=”OBJECT-TYPE”/>
  </imports>

  <typedefs>
```

```
  <typedef name=”DisplayString” basetype=”OctetString”>
  </typedef>
  <typedef name=”PhysAddress” basetype=”OctetString”>
  </typedef>
</typedefs>

<nodes>
  <node name=”mib-2” oid=”1.3.6.1.2.1”>
  </node>

  <!-- Deleted all groups except the Interfaces Group -->

  <node name=”interfaces” oid=”1.3.6.1.2.1.2”>
  </node>
  <scalar name=”ifNumber” oid=”1.3.6.1.2.1.2.1” status=”current”>
    <syntax>
      <type module=”” name=”Integer32”/>
    </syntax>
    <access>readonly</access>
    <description>
        The number of network interfaces (regardless of
        their current state) present on this system.
    </description>
  </scalar>
  <table name=”ifTable” oid=”1.3.6.1.2.1.2.2” status=”current”>
    <description>
        A list of interface entries.  The number of
        entries is given by the value of ifNumber.
    </description>
    <row name=”ifEntry” oid=”1.3.6.1.2.1.2.2.1” status=”current”>
      <linkage>
        <index module=”RFC1213-MIB” name=”ifIndex”/>
      </linkage>
      <description>
          An interface entry containing objects at the
          subnetwork layer and below for a particular
          interface.
      </description>
     <column name=”ifIndex” oid=”1.3.6.1.2.1.2.2.1.1” status=”current”>
        <syntax>
          <type module=”” name=”Integer32”/>
        </syntax>
        <access>readonly</access>
        <description>
            A unique value for each interface.  Its value
            ranges between 1 and the value of ifNumber.  The
            value for each interface must remain constant at
            least from one re-initialization of the entity’s
            network management system to the next re-
            initialization.
        </description>
      </column>
```

```
<column name="ifDescr" oid="1.3.6.1.2.1.2.2.1.2" status="current">
  <syntax>
    <typedef basetype="OctetString">
      <parent module="RFC1213-MIB" name="DisplayString"/>
      <range min="0" max="255"/>
    </typedef>
  </syntax>
  <access>readonly</access>
  <description>
      A textual string containing information about the
      interface.  This string should include the name of
      the manufacturer, the product name and the version
      of the hardware interface.
  </description>
</column>
<column name="ifType" oid="1.3.6.1.2.1.2.2.1.3" status="current">
  <syntax>
    <typedef basetype="Enumeration">
      <namednumber name="other" number="1"/>
      <namednumber name="regular1822" number="2"/>
      <namednumber name="hdh1822" number="3"/>
      <namednumber name="ddn-x25" number="4"/>
      <namednumber name="rfc877-x25" number="5"/>
      <namednumber name="ethernet-csmacd" number="6"/>
      <namednumber name="iso88023-csmacd" number="7"/>
      <namednumber name="iso88024-tokenBus" number="8"/>
      <namednumber name="iso88025-tokenRing" number="9"/>
      <namednumber name="iso88026-man" number="10"/>
      <namednumber name="starLan" number="11"/>
      <namednumber name="proteon-10Mbit" number="12"/>
      <namednumber name="proteon-80Mbit" number="13"/>
      <namednumber name="hyperchannel" number="14"/>
      <namednumber name="fddi" number="15"/>
      <namednumber name="lapb" number="16"/>
      <namednumber name="sdlc" number="17"/>
      <namednumber name="ds1" number="18"/>
      <namednumber name="e1" number="19"/>
      <namednumber name="basicISDN" number="20"/>
      <namednumber name="primaryISDN" number="21"/>
      <namednumber name="propPointToPointSerial" number="22"/>
      <namednumber name="ppp" number="23"/>
      <namednumber name="softwareLoopback" number="24"/>
      <namednumber name="eon" number="25"/>
      <namednumber name="ethernet-3Mbit" number="26"/>
      <namednumber name="nsip" number="27"/>
      <namednumber name="slip" number="28"/>
      <namednumber name="ultra" number="29"/>
      <namednumber name="ds3" number="30"/>
      <namednumber name="sip" number="31"/>
      <namednumber name="frame-relay" number="32"/>
    </typedef>
  </syntax>
```

```
          <access>readonly</access>
          <description>
              The type of interface, distinguished according to
              the physical/link protocol(s) immediately `below'
              the network layer in the protocol stack.
          </description>
        </column>
        <column name="ifMtu" oid="1.3.6.1.2.1.2.2.1.4" status="current">
          <syntax>
            <type module="" name="Integer32"/>
          </syntax>
          <access>readonly</access>
          <description>
              The size of the largest datagram which can be
              sent/received on the interface, specified in
              octets.  For interfaces that are used for
              transmitting network datagrams, this is the size
              of the largest network datagram that can be sent
              on the interface.
          </description>
        </column>
       <column name="ifSpeed" oid="1.3.6.1.2.1.2.2.1.5" status="current">
          <syntax>
            <type module="RFC1155-SMI" name="Gauge"/>
          </syntax>
          <access>readonly</access>
          <description>
              An estimate of the interface's current bandwidth
              in bits per second.  For interfaces which do not
              vary in bandwidth or for those where no accurate
              estimation can be made, this object should contain
              the nominal bandwidth.
          </description>
        </column>
                <column  name="ifPhysAddress"  oid="1.3.6.1.2.1.2.2.1.6"
status="current">
          <syntax>
            <type module="RFC1213-MIB" name="PhysAddress"/>
          </syntax>
          <access>readonly</access>
          <description>
              The interface's address at the protocol layer
              immediately `below' the network layer in the
              protocol stack.  For interfaces which do not have

              such an address (e.g., a serial line), this object
              should contain an octet string of zero length.
          </description>
        </column>
                <column  name="ifAdminStatus"  oid="1.3.6.1.2.1.2.2.1.7"
status="current">
          <syntax>
```

```
            <typedef basetype=”Enumeration”>
              <namednumber name=”up” number=”1”/>
              <namednumber name=”down” number=”2”/>
              <namednumber name=”testing” number=”3”/>
            </typedef>
          </syntax>
          <access>readwrite</access>
          <description>
              The desired state of the interface.  The
              testing(3) state indicates that no operational
              packets can be passed.
          </description>
        </column>
                  <column  name=”ifOperStatus”  oid=”1.3.6.1.2.1.2.2.1.8”
status=”current”>
          <syntax>
            <typedef basetype=”Enumeration”>
              <namednumber name=”up” number=”1”/>
              <namednumber name=”down” number=”2”/>
              <namednumber name=”testing” number=”3”/>
            </typedef>
          </syntax>
          <access>readonly</access>
          <description>
              The current operational state of the interface.
              The testing(3) state indicates that no operational
              packets can be passed.
          </description>
        </column>
                  <column  name=”ifLastChange”  oid=”1.3.6.1.2.1.2.2.1.9”
status=”current”>
          <syntax>
            <type module=”RFC1155-SMI” name=”TimeTicks”/>
          </syntax>
          <access>readonly</access>
          <description>
              The value of sysUpTime at the time the interface
              entered its current operational state.  If the
              current state was entered prior to the last re-
              initialization of the local network management
              subsystem, then this object contains a zero
              value.
          </description>
        </column>
                   <column  name=”ifInOctets”  oid=”1.3.6.1.2.1.2.2.1.10”
status=”current”>
          <syntax>
            <type module=”RFC1155-SMI” name=”Counter”/>
          </syntax>
          <access>readonly</access>
          <description>
              The total number of octets received on the
```

```
              interface, including framing characters.
          </description>
        </column>
                <column  name="ifInUcastPkts"  oid="1.3.6.1.2.1.2.2.1.11"
status="current">
          <syntax>
            <type module="RFC1155-SMI" name="Counter"/>
          </syntax>
          <access>readonly</access>
          <description>
              The number of subnetwork-unicast packets
              delivered to a higher-layer protocol.
          </description>
        </column>
                 <column  name="ifInNUcastPkts"  oid="1.3.6.1.2.1.2.2.1.12"
status="current">
          <syntax>
            <type module="RFC1155-SMI" name="Counter"/>
          </syntax>
          <access>readonly</access>
          <description>
              The number of non-unicast (i.e., subnetwork-
              broadcast or subnetwork-multicast) packets
              delivered to a higher-layer protocol.
          </description>
        </column>
                  <column  name="ifInDiscards"  oid="1.3.6.1.2.1.2.2.1.13"
status="current">
          <syntax>
            <type module="RFC1155-SMI" name="Counter"/>
          </syntax>
          <access>readonly</access>
          <description>
              The number of inbound packets which were chosen
              to be discarded even though no errors had been
              detected to prevent their being deliverable to a
              higher-layer protocol.  One possible reason for
              discarding such a packet could be to free up
              buffer space.
          </description>
        </column>
                    <column  name="ifInErrors"  oid="1.3.6.1.2.1.2.2.1.14"
status="current">
          <syntax>
            <type module="RFC1155-SMI" name="Counter"/>
          </syntax>
          <access>readonly</access>
          <description>
              The number of inbound packets that contained
              errors preventing them from being deliverable to a
              higher-layer protocol.
          </description>
```

```
        </column>
             <column  name=”ifInUnknownProtos”  oid=”1.3.6.1.2.1.2.2.1.15”
status=”current”>
          <syntax>
            <type module=”RFC1155-SMI” name=”Counter”/>
          </syntax>
          <access>readonly</access>
          <description>
              The number of packets received via the interface
              which were discarded because of an unknown or
              unsupported protocol.
          </description>
        </column>
                  <column  name=”ifOutOctets”  oid=”1.3.6.1.2.1.2.2.1.16”
status=”current”>
          <syntax>
            <type module=”RFC1155-SMI” name=”Counter”/>
          </syntax>
          <access>readonly</access>
          <description>
              The total number of octets transmitted out of the
              interface, including framing characters.
          </description>
        </column>
              <column  name=”ifOutUcastPkts”  oid=”1.3.6.1.2.1.2.2.1.17”
status=”current”>
          <syntax>
            <type module=”RFC1155-SMI” name=”Counter”/>
          </syntax>
          <access>readonly</access>
          <description>
              The total number of packets that higher-level
              protocols requested be transmitted to a
              subnetwork-unicast address, including those that
              were discarded or not sent.
          </description>
        </column>
             <column  name=”ifOutNUcastPkts”  oid=”1.3.6.1.2.1.2.2.1.18”
status=”current”>
          <syntax>
            <type module=”RFC1155-SMI” name=”Counter”/>
          </syntax>
          <access>readonly</access>
          <description>
              The total number of packets that higher-level
              protocols requested be transmitted to a non-
              unicast (i.e., a subnetwork-broadcast or
              subnetwork-multicast) address, including those
              that were discarded or not sent.
          </description>
        </column>
```

```
                <column  name="ifOutDiscards"  oid="1.3.6.1.2.1.2.2.1.19"
status="current">
          <syntax>
            <type module="RFC1155-SMI" name="Counter"/>
          </syntax>
          <access>readonly</access>
          <description>
              The number of outbound packets which were chosen

              to be discarded even though no errors had been
              detected to prevent their being transmitted.  One
              possible reason for discarding such a packet could
              be to free up buffer space.
          </description>
        </column>
                  <column  name="ifOutErrors"  oid="1.3.6.1.2.1.2.2.1.20"
status="current">
          <syntax>
            <type module="RFC1155-SMI" name="Counter"/>
          </syntax>
          <access>readonly</access>
          <description>
              The number of outbound packets that could not be
              transmitted because of errors.
          </description>
        </column>
                   <column   name="ifOutQLen"  oid="1.3.6.1.2.1.2.2.1.21"
status="current">
          <syntax>
            <type module="RFC1155-SMI" name="Gauge"/>
          </syntax>
          <access>readonly</access>
          <description>
              The length of the output packet queue (in
              packets).
          </description>
        </column>
                   <column  name="ifSpecific"  oid="1.3.6.1.2.1.2.2.1.22"
status="current">
          <syntax>
            <type module="" name="ObjectIdentifier"/>
          </syntax>
          <access>readonly</access>
          <description>
              A reference to MIB definitions specific to the
              particular media being used to realize the
              interface.  For example, if the interface is
              realized by an ethernet, then the value of this
              object refers to a document defining objects
              specific to ethernet.  If this information is not
              present, its value should be set to the OBJECT
              IDENTIFIER { 0 0 }, which is a syntatically valid
```

```
              object identifier, and any conformant
              implementation of ASN.1 and BER must be able to
              generate and recognize this value.
          </description>
        </column>
      </row>
    </table>

    <!-- Deleted all groups except the Interfaces Group -->

  </nodes>

</smi>
```

Appendix C

METAMODEL-LEVEL XML MAPPING OF A SIMPLE CIM CLASS

In this appendix, we reproduce “as is” an example of CIM-to-XML metamodel-level mapping given by the DMTF [64]. The purpose of this example is not to show *the* way to map the CIM class below in XML, but rather to illustrate the metamodel-level mapping philosophy of the DMTF. This mapping is used in Chapter 8.

The MOF syntax for the class is shown below:

```
[Abstract] class CIM_ManagedSystemElement
{
[MaxLen(64)] string Caption;
string Description;
[MappingStrings{"MIF.DMTF|ComponentID|001.5"}] datetime InstallDate;
string Name;
[Values{"OK","Error","Degraded","Unknown"}] string Status;
};
```

The corresponding XML mapping for this class is shown below:

```
<?xml version=”1.0” ?>

<!DOCTYPE CIM SYSTEM
“http://WBEM_TECRA_2/wbem/cim.dtd”>
<CIM CIMVERSION=”2.0”
DTDVERSION=”1.0” >
<CLASS>
<CLASSPATH>
```

```
<NAMESPACEPATH>
<HOST>WBEM_TECRA_2</HOST>
<NAMESPACE>
<NAMESPACENODE>ROOT</NAMESPACENODE>
<NAMESPACE>
<NAMESPACENODE>CIMV2</NAMESPACENODE>
</NAMESPACE>
</NAMESPACE>
</NAMESPACEPATH>
<CLASSNAME>CIM_ManagedSystemElement</CLASSNAME>
</CLASSPATH>
<QUALIFIER NAME=”Abstract”
LOCAL=”true” TYPE=”boolean”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE>TRUE</VALUE>
</QUALIFIER>
<PROPERTY NAME=”Caption” CLASSORIGIN=
“CIM_ManagedSystemElement”
LOCAL=”true”TYPE=”string”>
<QUALIFIER NAME=”CIMTYPE”
LOCAL=”true” TYPE=”string”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”ToSubclass”
TRANSLATABLE=”false”>
<VALUE>string</VALUE>
</QUALIFIER>
<QUALIFIER NAME=”MaxLen” LOCAL=”true”
TYPE=”sint32”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE>64</VALUE>
</QUALIFIER>
<QUALIFIER NAME=”read” LOCAL=”true”
TYPE=”boolean”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE>TRUE</VALUE>
</QUALIFIER>
</PROPERTY>
<PROPERTY NAME=”Description”
CLASSORIGIN=”CIM_ManagedSystemElement”
LOCAL=”true” TYPE=”string”>
<QUALIFIER NAME=”CIMTYPE”
LOCAL=”true” TYPE=”string”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”ToSubclass”
TRANSLATABLE=”false”>
<VALUE>string</VALUE>
```

```
</QUALIFIER>
<QUALIFIER NAME=”read” LOCAL=”true”
TYPE=”boolean”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE>TRUE</VALUE>
</QUALIFIER>
</PROPERTY>
<PROPERTY NAME=”InstallDate”
CLASSORIGIN=”CIM_ManagedSystemElement”
LOCAL=”true” TYPE=”datetime”>
<QUALIFIER NAME=”CIMTYPE”
LOCAL=”true” TYPE=”string”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”ToSubclass”
TRANSLATABLE=”false”>
<VALUE>datetime</VALUE>
</QUALIFIER>
<QUALIFIER NAME=”MappingStrings”
LOCAL=”true” TYPE=”string”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE.INDEXED INDEX
=”0”>MIF.DMTF|ComponentID|001.5</VALUE.INDEXED>
</QUALIFIER>
<QUALIFIER NAME=”read” LOCAL=”true”
TYPE=”boolean”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE>TRUE</VALUE>
</QUALIFIER>
</PROPERTY>
<PROPERTY NAME=”Name”
CLASSORIGIN=”CIM_ManagedSystemElement”
LOCAL=”true” TYPE=”string”>
<QUALIFIER NAME=”CIMTYPE”
LOCAL=”true” TYPE=”string”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”ToSubclass”
TRANSLATABLE=”false”>
<VALUE>string</VALUE>
</QUALIFIER>
<QUALIFIER NAME=”read” LOCAL=”true”
TYPE=”boolean”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE>TRUE</VALUE>
</QUALIFIER>
```

```
</PROPERTY>
<PROPERTY NAME=”Status”
CLASSORIGIN=”CIM_ManagedSystemElement”
LOCAL=”true” TYPE=”string”>
<QUALIFIER NAME=”CIMTYPE”
LOCAL=”true” TYPE=”string”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”ToSubclass”
TRANSLATABLE=”false”>
<VALUE>string</VALUE>
</QUALIFIER>
<QUALIFIER NAME=”read” LOCAL=”true”
TYPE=”boolean”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE>TRUE</VALUE>
</QUALIFIER>
<QUALIFIER NAME=”Values” LOCAL=”true”
TYPE=”string”
OVERRIDABLE=”EnableOverride”
TOSUBCLASS=”Restricted”
TRANSLATABLE=”false”>
<VALUE.INDEXED INDEX
=”0”>OK</VALUE.INDEXED>
<VALUE.INDEXED INDEX
=”1”>Error</VALUE.INDEXED>
<VALUE.INDEXED INDEX
=”2”>Degraded</VALUE.INDEXED>
<VALUE.INDEXED INDEX
=”3”>Unknown</VALUE.INDEXED>
</QUALIFIER>
</PROPERTY>
</CLASS>
</CIM>
```

Appendix D

REMOTE METHOD INVOCATION OF A CIM OBJECT

In this appendix, we reproduce “as is” an example of remote method invocation of a CIM object given by the DMTF [64, pp. 75–76]. Compared to the original document, we simply removed the HTTP header of the two messages. The purpose of this example is to illustrate the containment hierarchy of operations and method invocations as well as the use of the namespace. This XML document is used in Chapter 8.

Request message:

```
<?xml version="1.0" encoding="utf-8" ?>
<CIM CIMVERSION="2.0" DTDVERSION="2.0">
 <MESSAGE ID="87872" PROTOCOLVERSION="1.0">
  <SIMPLEREQ>
   <METHODCALL NAME="SetPowerState">
    <LOCALINSTANCEPATH>
     <LOCALNAMESPACEPATH>
      <NAMESPACE NAME="root"/>
      <NAMESPACE NAME="myNamespace"/>
     </LOCALNAMESPACEPATH>
     <INSTANCENAME CLASSNAME="MyDisk">
      <KEYBINDING NAME="C:"><KEYVALUE>C:</KEYVALUE></KEYBINDING>
     </INSTANCENAME>
    </LOCALINSTANCEPATH>
    <PARAMVALUE NAME="PowerState"><VALUE>1</VALUE></PARAMVALUE>
    <PARAMVALUE NAME="Time">
     <VALUE>00000001132312.000000:000</VALUE>
    </PARAMVALUE>
   </METHODCALL>
  </SIMPLEREQ>
 </MESSAGE>
</CIM>
```

Response message:

```
<?xml version="1.0" encoding="utf-8" ?>
 <CIM CIMVERSION="2.0" DTDVERSION="2.0">
  <MESSAGE ID="87872" PROTOCOLVERSION="1.0">
   <SIMPLERSP>
    <METHODRESPONSE NAME="SetPowerState">
     <RETURNVALUE>
      <VALUE>0</VALUE>
     </RETURNVALUE>
    </METHODRESPONSE>
   </SIMPLERSP>
  </MESSAGE>
 </CIM>
```

CURRICULUM VITAE

J.P. Martin-Flatin graduated as an *ingénieur généraliste* (U.S. equivalent: M.S. in electrical and mechanical engineering) from ECAM, Lyon, France in 1986. From 1996 to 2000, he worked as a research and teaching assistant in the group of Prof. J.P. Hubaux. While remaining in the same group, he was successively affiliated with the Electrical Engineering Dept., the Computer Science Dept., and the newly created Communication Systems Dept. From 1990 to 1996, he worked as a CS engineer with the European Centre for Medium-Range Weather Forecasts (ECMWF) in Reading, UK. His activities included network and systems management, software development, Web engineering, network security, and systems security. From 1988 to 1990, he worked as a CS engineer with Icare, Lyon, France on the geographic information system of the Greater Lyon. His activities centered on systems management and software development.

J.P. is a member of the IEEE Computer and Communications Societies, ACM, IRTF Network Management Research Group, and DMTF WBEM Interoperability Working Group. He is currently serving as a guest editor for the Journal of Network and Systems Management (special issue on Web-Based Management). He is a member of the Technical Program Committees of the three main events in his research field (IM, NOMS, and DSOM). He is a reviewer for most journals, conferences, and workshops in his field. In 1995, he participated in the activities of the IETF HTTP Working Group. He is listed as one of the contributors to the specification of HTTP/1.0 (RFC 1945).