JAVA-BASED INTELLIGENT NETWORK MONITORING

by

Haifeng Xi

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2000

Advisory Committee:

Professor John S. Baras, Chair
Professor Mark Austin
Professor Nick Roussopoulos

©Copyright by

Haifeng Xi

2000

ABSTRACT

Title of Thesis:	JAVA-BASED INTELLIGENT NETWORK MONITORING
Degree Candidate:	Haifeng Xi
Degree and year:	Master of Science, 2000
Thesis directed by:	Professor John S. Baras Department of Electrical and Computer Engineering

The increasing complexity and importance of communication networks have given rise to a steadily high demand for advanced network management tools. Network Management in general consists of two activities: monitoring and controlling. The monitoring part concerns observing and analyzing the status and behavior of the managed networks, and is therefore fundamental for network management. Unfortunately the existing network monitoring paradigms have some drawbacks that prevent it from satisfactory performance. One related problem is that these approaches are characterized by high centralization which puts almost all the computational burden on the management station. As a result, a huge amount of raw data have to be transferred from network elements to the central management station for further processing, causing heavy traffic, manager overload and long operations delay. Another issue that becomes increasingly noticeable is the absence of a mechanism for dynamic extensions to agent functionality.

In this work we take advantage of some unique features of the Java technology and present a framework for distributed and dynamic network monitoring. Specialized Java objects known as Intelligent Monitoring Objects, are delegated to a Java-based Extensible Management Server (JEMS), where they carry out encapsulated monitoring functionality upon management information collected locally from the underlying network device. We have built a proof-of-concept prototype system using the JEMS architecture and validated its effectiveness and flexibility compared with the traditional centralized network management systems.

ACKNOWLEDGEMENTS

I would like to express my deep regard for Dr. John S. Baras to whom I am extremely grateful. He allowed me the liberty of exploring my areas of interest, and the flexibility of writing the thesis off-campus. He provided me with the valuable opportunity to work on real-world problems with smart people from both his Center for Satellite and Hybrid Communication Networks and the industry.

I would like to thank Dr. Mark Austin and Dr. Nick Roussopoulos for consenting to be on my thesis committee while they are on very tight schedules.

I feel extremely indebted to Mr. Hongjun Li, my best friend, who was always available to discuss research issues, and to seek help on administrative stuffs while I am away from campus. I would also like to express my gratitude to Dr. George Mykoniatis for his advice and insights on object-oriented design.

This work was funded through NASA Cooperative Agreement NCC3-528, the Advanced Telecommunications/Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0002, National Science Foundation, Hughes Network Systems, and Northrop Grumman.

Most of all I would like to thank my wife without whom the completion of this work would not have been possible. She has been a source of continuous support and encouragement. Finally I would like to thank all my friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

This thesis introduces Java-based Intelligent Network Monitoring, an efficient approach to monitoring networked systems using the proposed Java-based extensible management server and various intelligent monitoring objects.

The increasing importance of communication networks has given rise to a high demand for advanced network management. A network management system handles problems related to the configurability, reliability, efficiency, security and accountability of the managed distributed computing environments. It is concerned with monitoring, analysis and control of network behaviors to ensure smooth network operations. Accurate and effective monitoring is fundamental and critical for network management, and is the focus of our work presented in this thesis.

In the mainstream network monitoring system, operational data are collected by hardwired instrumentation in network elements and stored in Management Information Bases (MIBs). For example, the MIB in an ATM switch can include predefined counters and gauges about various traffic statistics for virtual circuits, virtual paths and switch ports. This operational data is gathered, usually remotely over a network, by a central Network Management Station (NMS) using a network management protocol. The NMS presents the data to operations staff who are responsible for monitoring, analyzing and controlling the network. This centralized and static management paradigm does not scale for the size and complexity of emerging heterogeneous broadband networks, neither does it adapt to unforeseen management requirements. Therefore, new technologies are needed to decentralize management functions in a dynamic way.

Intelligent monitoring is the conception brought forward in this thesis to address the aforementioned problems. Management applications distribute or delegate monitoring objects to Java-based extensible management servers (JEMS) running at network elements. These objects will automate the monitoring and analysis of corresponding network devices. For example, a delegated object can monitor a MIB variable and compare its value against some pre-set thresholds to detect potential operations problems. Monitoring intelligence and responsibilities can thus be decentralized. Furthermore, when creating and manipulating a delegated object, the JEMS takes advantage of Java's dynamic class loading feature to download code over the network, which results in a highly adaptable monitoring server structure whose functionality can be dynamically extended.

Chapter Organization

Section 1.1 outlines network management/monitoring, and limitations of current network monitoring systems.

Section 1.2 summarizes the contributions of this thesis.

Section 1.3 briefly introduces intelligent network monitoring and its benefits.

Section 1.4 looks into several research efforts that related to our work.

Section 1.5 presents a roadmap of the remaining chapters in this thesis.

1.1 Network Management and Monitoring

Network management systems handle problems related to the configurability, reliability, efficiency, security and accountability of the managed distributed computing environments, and are concerned with monitoring and control of network behaviors to ensure smooth network operations. The network monitoring portion of network management is concerned with observing and analyzing the status and behavior of the managed network devices.

1.1.1 Components of Network Management System

A conventional network management system consists of two classes of components: *managers* and *element agents.* Figure 1 depicts a diagram of the

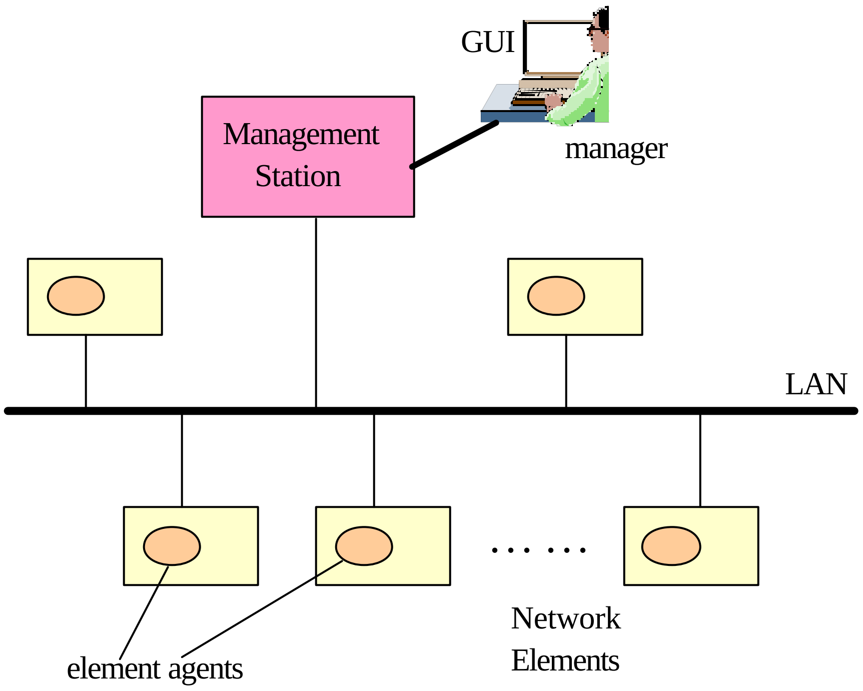

Figure 1: Components of typical Network Management Systems

organization of a typical network management system. Applications in the central management station assume the manager role, and execute with a GUI for human managers to perform certain monitoring functions. Element agents are server processes running in each involved manageable network entity. These agents collect device data, stores them in the Management Information Bases (MIBs), and support a management protocol, e.g., Simple Network Management Protocol (SNMP) [1,2]. Manager applications retrieve data from element agents by sending corresponding requests over the management protocol.

For example, the SNMP agent in an ATM switch collects information about the signaling protocol, the traffic status of virtual circuits etc. and stores it in a predefined MIB. A management application retrieves this data using the SNMP GetRequest command, processes and analyzes them, and then displays the result graphically.

1.1.2 Limitations of Current Systems

Current network monitoring/management systems favor a centralized framework where most of the monitoring intelligence and computation burdens are allocated to the manager applications executing at the central station. This establishes several barriers to effective network monitoring, especially for emerging high-speed networks. Several major problems are outlined in the following paragraphs.

Given the centralized allocation of management responsibilities, all the monitoring interactions and processing have to go through the management station, which becomes the bottleneck and single point of failure. This leads to a system that hardly scales up to large and complex networks.

Since manager applications can only interact with the network elements through low-level general-purpose interfaces, any non-trivial monitoring task requires huge volume of “raw” SNMP variables being transferred to the management station, which is known as the *micro-management* problem. Micro-management results in high communication overheads, and significant operation delays if the managed network is wireless or satellite-based.

The set of services offered by the element agents is fixed, strictly defined by standards, and is accessible through interfaces that are statically defined and implemented. This service set cannot be modified or extended on the fly without the recompilation, reinstallation, and reinstantiation of the server process. This rigid and static agent structure hinders the development of effective monitoring systems in two ways: (1) improvement of the agent usually involves high-cost activities and may harm system availability; (2) it does not provide any mechanism to differentiate between and take advantage of the capabilities of different types of devices.

1.2 Contributions

The major contributions presented in this thesis include:

Java-based Extensible Management Server (JEMS). A model that supports intelligent, i.e., distributed and dynamic, network monitoring. JEMS runs as a server process in the managed network element and consists of: (1) the Remote Delegation Interface (RDI) through which management applications can remotely delegate Java objects, exchange information with these objects and control their execution; (2) a runtime environment that implements the RDI.

Intelligent Monitoring Objects (IMOs). Specialized Java objects that perform network monitoring functions. IMOs are distributed to JEMS at the network devices where the managed resources are located. We have categorized common monitoring functions and encapsulated them into corresponding IMOs. IMOs are implemented in such a way that they work closely with the underlying JEMS to provide for an intelligent monitoring system.

1.3 Intelligent Network Monitoring

The approach of intelligent network monitoring is to dynamically distribute monitoring functionality, in the form of IMOs, to JEMS at the devices where the managed resources are located. Specifically speaking, intelligent monitoring means two things:

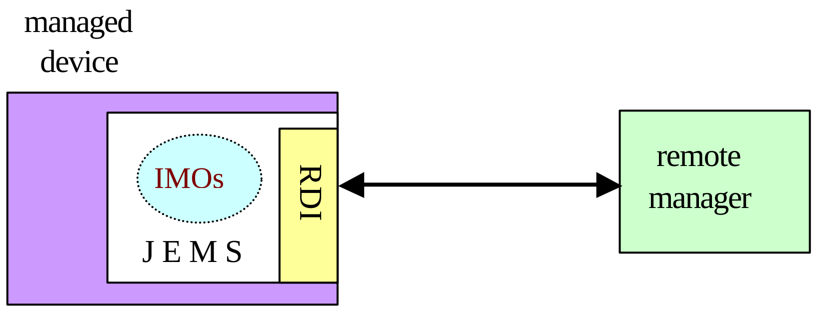

Figure 2: IMOs delegated to JEMS

First, manager applications can distribute monitoring intelligence to the managed network element. Instead of bringing data from the devices to the central station, parts of the monitoring applications themselves, encapsulated in various IMOs, are actually running in the managed devices (Figure 2). The manager host and the network as a whole can then be relieved from the bottleneck and the micro-management problems.

Secondly, the network element's agent functionality can be dynamically modified or extended. Through the remote delegation interface (RDI), manager applications can choose to distribute/delete whatever IMOs to/from the device, at whatever time they like. Furthermore, the code of a Java class need not be available beforehand to the JEMS when a corresponding object is instantiated upon a delegation request from the manager. JEMS can download class code over the network and link it to the runtime system on demand, which makes it truly dynamically extensible.

1.3.1 Advantages over Current Systems

Intelligent network monitoring allows for dynamic extensions to monitoring functionality as the network evolves. For instance, if a new monitoring requirement is identified after the system is up and running, we can create a corresponding IMO that encapsulates the functionality and delegate it to the JEMS, without having to bring the system offline.

By recognizing and taking advantage of the difference in resource availability of various network devices, manager applications can have flexible and efficient usage of these resources. For devices with a lot of resources (e.g., memory and CPU processing power), the manager can delegate large numbers of IMOs to them, making full use of local computing to reduce management traffic overhead and delay. This is a great advantage over traditional static systems, where the size and funtionality of network agents are fixed once they start running, and extra device resources can not be used to improve the efficiency of network monitoring. For devices with relatively few resources, the manager can always switch back to the traditional polling-based paradigm.

Similarly, remote managers can dynamically adapt to changes in the availability of network and computing resources. For example, when the network is overloaded, manager applications can encapsulate related monitoring functions in some IMOs that are distributed to the JEMS and analyze data locally at the devices, relieving the network from the traffic overloading that frequent polling would otherwise have worsened.

1.4 Related Work

There are several research efforts that are related to our work in one way or another. We will try to identify the strength and weakness of each of them and to spell out their distinctions from our system.

1.4.1 Management by Delegation

Management by Delegation (MbD) [19] is one of the earliest efforts towards decentralization and increased flexibility of management functionality, and is probably the most successful one. Its management architecture includes a management protocol, device agents, and an elastic process run-time support on each device. Instead of exchanging basic client-server messages, the management station can specify a task by packing into a program a set of agent commands and sending it to the devices involved, thus delegating to them the actual execution of the task. Such execution is completely asynchronous, enabling the management station to perform other tasks in the mean time and introducing a higher degree of parallelism in the whole management architecture. MbD greatly influenced later research and exploration along this direction [12,13,14].

However, there are two disadvantages with MbD that probably have affected its application: (1) Since the work was done before 1996, when the Java platform was not so widely recognized and deployed by industry as it is today, the proof-of-concept MbD system was implemented with a proprietary server environment written in the C programming language – we hardly see any working systems that are built upon this proprietary environment. (2) The MbD server environment is so comprehensive and complicated that it can turn out to be an "overkill" in most real-world applications.

Still, we must give credit to MbD because it can be considered a precursor of the ideas discussed here. The major difference is that we have adopted the standard Java platform and, from the very beginning, aimed to build a portable, simple, yet powerful framework that can be easily understood, implemented and enhanced. Because of the wide installation base and intrinsic portability of the Java platform, we expect to see quick adoption of our system (or a derived version of it) in some relevant real-world applications.

1.4.2 Mobile Agents

Using mobile agents in decentralized and intelligent network management

[13] is a great leap from client-server based management pattern. The proposal is that an intelligent agent containing management code can traverse in the managed network from node to node, autonomously retaining the state of its computation whenever it moved to a different node.

There are two potential problems with this method: (1) It proposed a change to network management paradigm that is so radical that even the authors themselves realized that "further validation with quantitative data" would be necessary to prove its effectiveness. (2) Their proposed agents were written in specialized agent-oriented scripting languages such as Telescript [22] and Agent Tcl [23], which are not available or supported on many platforms, posing serious portability problems.

In contrast, our system still retains a client-server architecture, and assumes a management server in each device concerned. Therefore, while IMOs provide similar intelligent monitoring functions as their mobile agent counterparts, their behaviors are much easier to understand and anticipate, making them more straightforward to integrate and co-exist with traditional systems.

1.4.3 Web-based Network Management

We are by no means the first people thinking of using Java technology in network management [14,15,16]. But so far we have only seen efforts focused on so-called Web-based Network Management, which addresses the problem of integrating manager applications with the Web using Java applets. This is a well-justified idea that attempts to provide a cost effective way of providing uniform management services to managers and potential customers through such common client-side interface as Web browsers.

Instead, we have used Java for a totally different purpose, which is not to facilitate client-side presentation or Web integration, but to use Java's native support for distributed computing, remote class downloading and object serialization to implement dynamic and intelligent network monitoring. However, it makes perfect sense to consider including Web-based front-ends into our future enhanced systems.

1.5 Thesis Roadmap

We will begin each chapter with a brief discussion of the main challenges and solutions presented in the chapter, followed by an outline of the organization of sections in the chapter.

Chapter 2 introduces network management and network monitoring. It illustrates the structure of conventional systems and points out its limitations. It also summarizes the most commonly used monitoring functions.

Chapter 3 presents the JEMS architecture. It defines the RDI, explains why Java is chosen as the foundation technology, and then examines the design and implementation of the JEMS.

Chapter 4 describes intelligent monitoring objects. It explains the design of different types of objects, and shows, via examples, how they work within the JEMS framework to provide for an intelligent monitoring system.

Chapter 5 evaluates intelligent monitoring and demonstrates its merits. It contrasts intelligent monitoring with centralized monitoring approaches within some simple but typical scenarios.

Chapter 6 draws conclusions and points out several future work directions.

Chapter 2 Network Monitoring

The main goal of network management systems is to ensure the quality of services (QoS) that the network provides. To achieve this, network managers must monitor and control the connected elements in the network.

The network monitoring portion of network management is concerned with observing and analyzing the status and behavior of the network devices that make up the configuration to be managed. Accurate and effective monitoring is therefore fundamental and critical for the implementation of various network management functions.

Chapter Organization

Section 2.1 outlines the architecture of current network management systems.

Section 2.2 examines SNMP, the dominating network management protocol.

Section 2.3 introduces network management functions.

Section 2.4 summarizes network monitoring functional requirements.

2.1 Network Management Systems Architecture

Figure 1 depicted the architecture of a conventional network management system. In a managed network element (e.g. router, switch or host), an *agent*, which is typically a small-footprint program running as a daemon process, collects device information in a predefined manner and stores them in the MIB. Management applications execute on some dedicated workstations located in the network operation center (NOC) and interface with human operators. These applications perform specific management functions and assume a *manager* role. They use a management protocol to periodically poll the agents in the managed network devices, requesting data of interest. Information retrieved from agents is some raw data (e.g., counters or gauges) and managers always have to perform certain amount of aggregating computation, for instance, figuring out min/max values, averages, variances etc., before any meaningful presentations can be forwarded to the NOC operators.

By bringing all the low-level data to the management stations, where they are further processed, traditional network management systems assume a centralized client-server paradigm, which poses some serious limitations that were presented in Section 1.1.2.

2.2 Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) [1,2] is a prevailing network management protocol designed and standardized by the IETF to provide for remote monitoring of IP-based networked devices. It is the most widely used protocol in current network management systems. Five types of protocol data units (PDUs) are defined for SNMP: three of them deal with reading data (GetRequest, GetNextRequest and GetResponse), one deals with setting data (SetRequest), and the last one, Trap, is used for monitoring network events such as device start-ups or shut-downs. In addition to the protocol itself, there are three other important components in an SNMP-based system: MIB, agent and manager.

Management Information Base (MIB) represents the information model of the managed device defined as a collection of variables, each of which has a name and syntax. For example, the Internet standard RFC1213 [18] defines the MIB for IP host management. In RFC1213, for instance, the variable that keeps record of the number of IP packets received is identified by the string name iso.org.dod.internet.mgmt.mib-2.ip.ipInReceives, and has the syntax as a Counter. Managers should have access to the definition files of the MIBs in those element agents, so that they know what data are available from the agents. Instantiated MIBs are organized as static trees with variable values stored at tree leaves. To retrieve the value of a specific MIB variable, a GetRequest command needs to provide the identity of the variable, which is derived from its name.

An SNMP agent runs as a server process in the managed network device, maintains an instantiation of the corresponding MIB, waits for and responds to SNMP PDUs from managers. Almost all the major internetwork hardware vendors provide SNMP agents with their products. For example, in a UNIX workstation, there is usually a daemon process, snmpd, running as the SNMP agent for the host.

SNMP managers are user applications that perform management functions by exchanging SNMP PDUs with SNMP agents in managed network elements.

Using SNMP to retrieve data from network devices is only the very first step to network management. In this sense, “SNMP” is quite a misleading name since the protocol itself supports nothing more than remote access to primitive device information. It is based on the functional level processing of such primitive information in the central management stations that the core of a network management system is built up.

2.3 Network Management Functions

There are various functional level requirements of network management. The most famous and frequently cited categorization is those five functional areas defined by the ISO, known as configuration, fault, performance, security and accounting management [1,20].

Among these five areas, fault and performance management are of the greatest interest to us, because they address the problems of system availability and QoS guarantee of the managed network. Once the network is initialized, configuration management is mainly used as a reconfiguration service by the fault, performance and security management to carry out their respective control operations. Security and accounting management are often handled by departments other than NOC because they involve a lot of specialized knowledge and procedures, such as authentication and encryption (security management), and pricing and billing (accounting management).

2.3.1 Network Management Functional Areas

Configuration Management

Modern communication systems are composed of physical and logical subsystems that can be configured to perform many different tasks. The same device, for example, can be configured to act either as a gateway or as an end system node, or both. Once the operator decides how he intends to use the device, he can choose to set values for the appropriate set of attributes associated with the device.

Configuration management is the aspect of network management which embodies the functionality to assign that set of attributes to the device. It concerns with initializing a network and gracefully shutting down part or all of it. It is also concerned with maintaining, adding, and updating the relationships among components and the status of components themselves during network operations. While the network is in operation, configuration management provides the ability to reconfigure the network in response to performance evaluation or in support of fault correction or security checks.

Fault Management

To maintain proper operation of a complex computer network, care must be taken so that the system as a whole, and each individual component, is in acceptable working order in presence of network faults, abnormal conditions requiring management attention to repair. Fault management is that aspect of network management which attends to these concerns.

The first and essential requirement of fault management is to detect the existence of a potential fault as quickly as possible, which demands continuous monitoring (i.e., observation and analysis) of relevant network operation statistics.

Once a potential fault is detected, a corresponding alarm is fired, which triggers the fault management module to identify or isolate the fault, i.e., to find out the root cause of the abnormality, if any. This process can be very difficult and often involves correlation of multiple alarms and even various tests, such as connectivity test, protocol integrity test and so on.

The result of fault identification could be: (1) no fault really exists; (2) the so-called fault is actually a performance degradation, and therefore should be further handled by performance management; or (3) there really is a fault and its root cause has been identified. In the last case, fault management will try to correct the fault. It can automatically reconfigure or modify the network in such a way that the impact on performance without the failed component(s) is minimized. Or it will notify the operator to repair or replace the failed components to restore the network to its initial state.

Performance Management

Whereas fault management is concerned with whether all or part of the network is working, performance management is concerned with how well the network or its parts are working, or the quality and effectiveness of network communications.

Similar to fault management, performance management comprises two broad functional categories – network monitoring and network control. Monitoring is the process of tracking and analyzing activities on the network. The controlling part enables performance management to make adjustments or reconfiguration to improve network performance. Some of the performance issues of concern are:

Is the link capacity under- or over-utilized?

Has throughput been reduced to an unacceptable level?

Are there any bottlenecks?

To deal with these concerns, the network operator must: (1) select a set of resource attributes to be monitored in order to assess performance levels, for example, utilization, throughput, rejection ratio of connection requests etc.; (2) associate appropriate metrics and values with relevant attributes as references of different levels of performance, for instance, one or more threshold values could be associated with an attribute; (3) continuously update the values of those indicator attributes and check them against the reference values. If the value of an attribute crosses a pre-defined threshold, a performance alarm could be fired, so that the performance management may step in and take corrective or preventive actions to keep the network working at an acceptable performance level. For example, if a link is or will be over-utilized, some traffic can then be re-routed through other switch ports for network traffic balancing.

Security Management

Security management is responsible for providing all the security related features such as authorized access, authentication and encryption. It maintains and distributes passwords and other authorization or access-control information, and generates, distributes and manages encryption keys. Security management is also concerned with monitoring and controlling access to computer networks and the network management information obtained from network nodes.

Accounting Management

This functional area is responsible for keeping a record of the usage of network resources by the network users. Each user's usage must be monitored and recorded, and the billing information updated accordingly. Billing information should be sent to the customers regularly.

2.3.2 Network Monitoring and Control

Orthogonal to the partition of network management into five functional areas, we can divide network management into two logical components, i.e., *network monitoring* and *network control*. Actually, each of the five functional areas examined in the last section can be roughly divided into these two portions.

Network Monitoring

Network monitoring further involves two steps: *observation* and *analysis*:

Observation. The process of maintaining up-to-date values of a set of indicators, which are some resource properties whose values are used to measure or evaluate the working status of a certain aspect of network functionality. *What indicators to observe* and *how to compute their values* are the two questions we have to answer.

Analysis. The process of detecting abnormal or deviating conditions based on observations made on the chosen set of indicators. The problems to address in this part are: *how to detect abnormality* (usually by comparing the indicators' values with some pre-specified "norms"), and *how to deal with it once detected* (generation of alarms is a common solution).

For example, security monitoring concerns observing and analyzing user access to computer networks in order to detect erroneous, illegal or malicious user operations that might compromise network security. Performance monitoring concerns observing and analyzing performance-related indicators, such as utility, throughput, availability etc., trying to detect performance degradations. Fault monitoring concerns observing and analyzing fault-related indicators (a.k.a. symptoms) in an effort to detect potential faults.

Network Control

Network control completes the other half of the management cycle by providing managers with the ability to modify or reconfigure certain parts of the network in order to restore it back to an acceptable working level when some abnormalities are detected and reported by network monitoring. Note that, because of the existence of uncertainty in observation, the reported abnormality has to be identified before any control actions can be taken. This identification process often involves complex global analyzing techniques such as event correlation [21], which is not considered part of network monitoring in this thesis.

2.4 Network Monitoring Functions

In this section, we look into details of network monitoring functions and try to answer those questions raised in the last section.

Note that our discussion is based on performance and fault monitoring only, because they comprise a significant portion of overall network monitoring activities and are where network QoS problems are addressed. Also note that the boundary between performance indicators and fault symptoms is being constantly blurred. For example, an unusually high packet retransmission rate could either be a performance indicator if it is caused by link over-utilization, or a fault symptom if it is due to an out-of-sync physical link that has a high bit error rate. Therefore it is not unusual to see a trend in the network management community to treat performance problems as "soft" faults. In this thesis, unless otherwise specified, network monitoring means the broader sense "performance-fault monitoring."

2.4.1 Network Observation

This section answers the questions asked in Section 2.3.2 about network observation, i.e., what types of indicators to observe, and how their values are computed.

Indicator Types

Most indicators that are useful for network monitoring fall into two categories: rate-oriented and ratio-oriented. Rate-oriented indicators reflect the varying speed of some underlying network attributes. A ratio-oriented indicator represents the proportional relationship between two quantities, usually in terms of percentage. Because of the inherent statistical nature of these indicators, they are sometimes further processed to generate some corresponding *statistics*, based on which analysis is finally carried out. Table 1 gives a breakdown of major indicators in each category, and the most commonly used statistics.

	Rate-Oriented Indicators
Throughput	The rate (count per unit time) at which some network events occurs, e.g., packet transfers, transactions.
Error Rate	The rate at which some errors occur, e.g., retransmission.
	Ratio-Oriented Indicators
Utilization	The percentage of the theoretical capacity of a resource (e.g., transmission line) that is being used.
Availability	The percentage of time that a network system, a component, or a software module is available for a user.
Accuracy	The percentage of time that no errors occur in the transmission and delivery of information.
	Indicator Statistics
Average	The average of an indicator over a specified sample size; various averaging methods could be used, e.g., smoothing average.
Variance	The variance of an indicator over a specified sample size.
Covariance	The covariance of two indicators over a given sample size.

Table 1: Indicator and Statistic Types

Note that, for fault monitoring, some other special indicators are very useful and must be monitored, among them are those SNMP traps representing the up/down status of system hardware such as network interface cards (NICs), communication links, switch ports and so on.

Indicator/Statistics Computation

In SNMP MIBs, there are only such primitive data types as counter, gauge and time ticker. The values of indicators and their statistics have to be computed based on these raw data.

(1) Rate-Oriented Indicators.

$$rate = D(value\, of\ \ SNMP\, variable)/Dt$$

To calculate the value of a specific rate-oriented indicator, therefore, the relevant SNMP variable and the time interval Dt have to be specified. For example, to calculate the error rate of received IP packets, we need to do such a calculation:

$$error\ rate = D(ipInReceives) / Dt$$

The value of the SNMP variable $ipInReceives$ (which is a counter) needs to be polled at the beginning and the end of the time interval respectively to obtain the difference $D(ipInReceives)$. To make the value up-to-date, the computation is performed every Dt seconds. The precision of this error rate depends on the length of Dt and how frequently the SNMP-agent updates the value of $ipInReceives$.

(2) Ratio-Oriented Indicators.

$$ratio = \frac{D(value\#i)}{D(value\#1) + D(value\#2) + \cdots + D(value\#k)}$$

To calculate the value of a ratio-oriented indicator, k SNMP variables and a computation cycle Dt have to be specified. For example, to calculate the call admission ratio (a form of availability) for a virtual circuit in the Fore ATM switch, during a time interval of Dt, the following formula is used:

$$\frac{\Delta(q2931CallsCompletions)}{\Delta(q2931CallsCompletions) + \Delta(q2931CallsRejections) + \Delta(q2931CallsFailures)}$$

Each of the three SNMP counter variables involved (defined in Fore-Switch-MIB) has to be polled twice, once at the beginning, the other at the end of the computation cycle. The ratio value is updated every Dt seconds.

(3) Indicator Statistics.

To calculate a statistic from an indicator, we need to have the sample size S, which is usually given by: $S = DT / Dt$ (where DT is the sampling period, and Dt is the updating cycle of the indicator value). For example, if we want to obtain the *average* value of the call admission ratio during a sampling period of DT, then we need to do the following computation:

$$average = \frac{1}{S} \times \left(\sum_{i=1}^{S} call\_admission\_ratio_i \right)$$

2.4.2 Network Analysis

Abnormality Detection

Abnormality detection is usually accomplished by comparing the values of indicators or their statistics to some pre-specified normal values. The simplest form of normal values are thresholds. When a threshold is crossed (from below or from above, as applied to different situations), an abnormal condition is considered to have occurred or to be occurring. More complex norms can be a set of values that comprise a pattern. When the degree to which the indicator/statistics values match this pattern has increased a predefined threshold, an abnormal condition is supposed to be there.

Those normal values are usually obtained beforehand through learning processes, which comprise a big research area in network management. Actually, one of the most challenging and controversial problems in network management is how to correctly interpret the meaning of those many indicators. This is not the major concern of this thesis, though.

Abnormality Reporting

Once a network abnormality is detected, a corresponding alarm is generated and sent to the network control module, where further (global) analysis and control actions are taken. Generally, an alarm contains such information as the time when it was created, the name of the alarm, the condition under which it was fired and so on.

Chapter 3 Java-based Extensible Management Server

In this thesis, we propose to build intelligent network monitoring systems to address some of the problems existing in current systems. As is suggested by our work, management applications distribute or delegate Intelligent Monitoring Objects (IMOs) to Java-based Extensible Management Servers (JEMS) running at network elements. JEMS is actually a Java-based element agent whose functionality can be extended by dynamically delegating various IMOs to it. IMOs are Java objects that automate the monitoring of corresponding network devices.

This chapter and the next chapter discuss the design and implementation of the JEMS and IMOs respectively.

Chapter Organization

Section 3.1 examines JEMS's Remote Delegation Interface (RDI).

Section 3.2 explains why Java is chosen as the platform for JEMS.

Section 3.3 explores the design and implementation of JEMS.

3.1 Remote Delegation Interface

Contrary to an SNMP agent, which has a fixed set of services, JEMS is an object-oriented element agent that provides extensible management services. Such extensibility comes from its Remote Delegation Interface (RDI) through which manager applications can dynamically delegate, remotely control and communicate with Java objects that perform monitoring functions in the managed network device. Since these objects are the entities that actually provide the functionality of the management server, and they can be dynamically downloaded to and created at the network element, we say that the management server is dynamically extensible.

Section 3.1.1 formally defines RDI in terms of delegation operations, and Section 3.1.2 presents a more detailed look at the RDI and how remote managers interact with JEMS through the RDI.

3.1.1 Formal Definition

An object-oriented process $P \equiv < C, S >$ consists of a code set C and a process status set S. $C \equiv \{c_1, c_2, ..., c_k\}$ is a set of class codes that P can execute, where c_i represents the code segment for a certain class. The process status $S \equiv \{o_1, o_2, ..., o_m\}$ is defined by the set of all the objects in the process. An object is defined by $o_i = (c^i, s^i)$, where $c^i \in C$ is the class code associated with object o_i, and s^i is the status of the object, such as the values of its data members and the execution state of its associated code c^i. Note that m is usually larger than k.

The JEMS process P_J is characterized by two dynamic sets of C and S, which can be modified via remote invocations of a set of *delegation operations* in the RDI. These operations allow manager applications to (1) extend the functionality of JEMS by delegating objects to it, (2) remotely control the execution of these objects, and (3) communicate with these objects.

Extensibility Operations

RDI_create operation incorporates a new object $o = (c, s)$ into $P_J \equiv < C, S >$.

$< C, S >, o \mapsto < C \cup \{c\},\ S \cup \{o\} >$

RDI_delete operation deletes an object $o = (c, s)$ from P_J.

$< C, S >, o \mapsto < C,\ S - \{o\} >$

Control Operations

RDI_disable operation suspends the functioning of object $o = (c, s)$.

$< C, S > \mapsto < C,\ S - \{o\} \cup \{(c, disabled)\} >$

RDI_enable operation starts or resumes the functioning of object *o*.

$< C, S > \mapsto < C,\ S - \{o\} \cup \{(c, running)\} >$

Communication Operations

RDI_set operation changes values of attributes of an object $o = (c, s)$.

$< C, S > \mapsto < C,\ S - \{o\} \cup \{(c, s')\} >$

RDI_get operation returns the value of some object attributes.

This operation does not change the state of relevant objects.

3.1.2 A Closer Look

RDI operations are summarized in Table 2 and illustrated in Figure 3:

RDI Operations
RDI_create (className, objName);
RDI_delete (objName);
RDI_enable (objName);
RDI_disable (objName);
RDI_set (objName, attrName, attrValue);
RDI_get (objName, attrName);

Table 2: Remote Delegation Interface

(1) Creation of an object in the JEMS is requested, using RDI_create

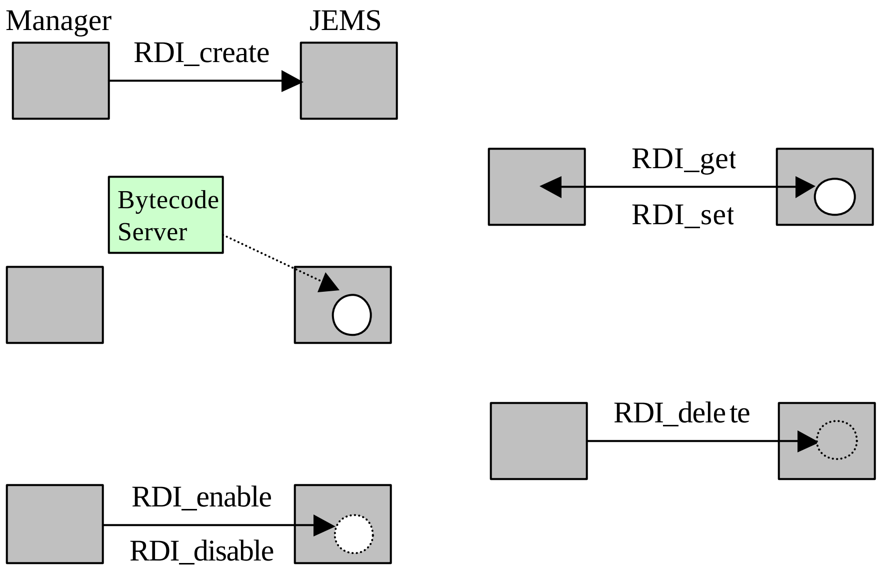

Figure 3: RDI Operations illustrated

operation. className specifies the name of the Java class from which the object is to be instantiated. objName is the name that will be used to identify the new object.

(2) The code for class className is downloaded over the network (if not already locally available) and dynamically linked to the runtime system, and a new

object is instantiated from the class.

(3) The manager can suspend and resume the functioning of the object, using RDI_disable and RDI_enable.

(4) The remote manager communicates with an enabled object using RDI_set and RDI_get operations.

(5) The manager removes an object using RDI_delete operation.

Note that the entities invoking a RDI operation may be either remote or local to the network device where the JEMS is running. We now examine each of these operation categories in more detail.

Extensibility Operations

Using RDI_create, a remote manager process requests that an object be incorporated into the management server P_J. The transfer of the class code from the bytecode server to the JEMS is performed by a class loader using Java's dynamic class loading feature. For an RDI_create to succeed, the following actions must be completed:

The class code must be checked to make sure that it is a legal Java class.

The new object and its code must fit within the resources available in P_J.

These two actions are performed implicitly by the underlying Java runtime environment, without any explicit application intervention. When an object is successfully instantiated, it has a unique name specified by the parameter objName. This name identifies the object and is later used to control and communicate with it.

A remote manager can delete or remove a delegated object using RDI_delete.

For an RDI_delete to succeed, the objName parameter must refer to a valid object.

Control Operations

Control operations allow remote managers to suspend or resume the functioning of an object. Any managed object, when first instantiated in P_J, is inactive before it is enabled by the remote manager through an RDI_enable

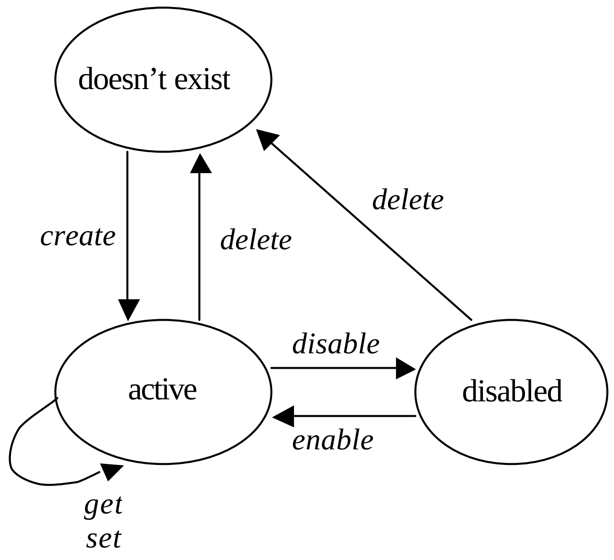

Figure 4: Life-cycle of a delegated Object

operation. An inactive or disabled object does not perform its normal monitoring function nor is it allowed to communicate with remote managers, until it is turned on and becomes an active object via an RDI_enable operation. An active object can be turned off at any time through an RDI_disable operation.

Communication Operations

RDI supports communications between a remote manager and delegated objects by allowing the manager to get or set the value of certain object attributes.

Using the RDI_set operation, the remote manager can change the attribute values of specific objects, thus changing the behavior of these objects. The manager process can get back information about delegated objects by retrieving attribute values from them using the RDI_get operation.

Figure 4 is a state diagram that depicts the life-cycle of a delegated object.

3.2 Why Java?

Taken individually, the characteristics of Java can be found in a variety of software development platforms. What's completely new is the manner in which Java technology and its runtime environment have combined them to produce a flexible and powerful programming system. Almost no other object-oriented programming languages or computing platforms provide all at once the following features, which are needed by the JEMS design:

Cross-platform Compatibility. In the proposed intelligent network monitoring framework, JEMSes can be running on UNIX or NT workstations, or IBM mainframes. Manager applications can run on different desktop environments such as UNIX, Linux, Windows or MacOS. Without a common platform, we have to write code with the same or similar functions for each of these platforms separately, which is very costly, time-consuming and error-prone. Java's architecture-neutral and portable ability makes it the an ideal platform base for a hybrid system like our network monitoring framework.

Distriubted Computing. The interactions between manager applications and JEMSes require distributed computing support from the programming language. Java provides native distributed programming API through Remote Method Invocation (RMI). Compared with other distributed programming platforms such as

DCOM or CORBA, Java RMI is the easiest to learn and use. Later you will see that it is very straightforward to implement the RDI with RMI.

Code on Demand. The dynamic feature of our framework requires support for code-on-demand (CoD) paradigm, i.e., objects in JEMS can download and link on-the-fly the code from some class server to perform a given task. Traditionally, CoD is only supported by mobile code languages (MCL), such as Telescript [22] and Agent Tcl [23], for mobile agent programming. With the dynamic class loading and linking ability, Java is actually a weak MCL that directly supports CoD.

Section 3.2.1 gives basic ideas about Java as both a language and a platform, and explains how it provides cross-platform compatibility. Section 3.2.2 introduces Java classes and interfaces, which are fundamental concepts repeatedly referred to in this thesis. Section 3.2.3 explains how Java provides distributed computing support via RMI, and Section 3.2.4 discusses a most important Java feature – dynamic class loading.

3.2.1 The Language and Platform

The Java Programming Language

Java is a high-level programming language that is object-oriented, interpreted, architecture-neutral and portable, distributed, dynamic, and secure. Each of the preceding buzzwords is explained in The Java Language Environment white paper [7].

Java is unusual in that each Java program is both compiled and interpreted. With a compiler, a Java class is translated into an intermediate language called Java bytecodes — the platform-independent codes interpreted by the Java interpreter. You can think of Java bytecodes as the machine code instructions for a Java Virtual Machine (JVM). Every Java interpreter is an implementation of the JVM.

Java bytecodes help make “write once, run anywhere” possible. You can compile your Java program into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the JVM. For example, the same Java program can run on Solaris, Windows NT, and Macintosh.

The Java Platform

The Java platform [8] differs from most other platforms in that it is a

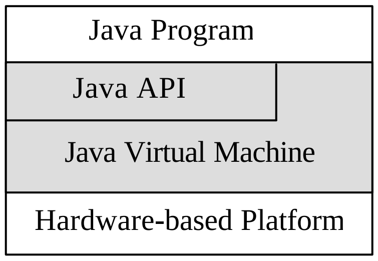

Figure 5: The Java Platform

software-only platform that runs on top of other hardware-based platforms. Most other platforms are described as a combination of hardware and operating system.

The Java platform has two components: the JVM and the Java Application Programming Interface (Java API) [9]. JVM is the base for the Java platform and is ported onto various hardware-based platforms. The Java API is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets. The Java API is grouped into packages of related components. The following figure depicts a Java program, such as an application or applet, that is running on the Java platform. As the figure shows, the

Java API and Virtual Machine insulates the Java program from hardware dependencies.

As a platform-independent environment, Java can be a bit slower than native code. However, smart compilers, well-tuned interpreters, and just-in-time compilers can bring Java's performance close to that of native code without threatening portability.

3.2.2 Classes and Interfaces

Classes and Objects

A *class* is a software construct that defines the data (state) and methods (behavior) of the specific concrete objects that are subsequently constructed from that class. In Java terminology, a class is built out of members, which are either attributes or methods. Attributes are the data for the class. Methods are the sequences of statements that operate on the data. Attributes are normally specific to an object — that is, every object constructed from the class definition will have its own copy of the attribute. Such attributes are known as instance variables. Similarly, methods are also normally declared to operate on the instance variables of the class, and are thus known as instance methods.

A class in and of itself is not an object. A class is like a blueprint that defines how an object will look and behave when the object is created or instantiated from the specification declared by the class. You obtain concrete objects by instantiating a previously defined class.

Subclasses are the mechanism by which new and enhanced objects can be defined in terms of existing objects. Subclasses enable you to use existing code that has already been developed and, much more important, tested, for a more generic case. You override the parts of the class you need for your specific behavior. Thus, subclasses gain you reuse of existing code — you save on design, development, and testing. Java implements what is known as a single-inheritance model: a new class can subclass (**extend**, in Java terminology) only one other class.

Java Interfaces

Interfaces were introduced to Java to enhance Java's single-inheritance model. An interface could be thought of as a pure abstract class. It allows the programmer to establish the form for a class: method names, argument lists and return types, but no instance variables or implementation code. An interface says: "This is what all classes that implement this particular interface will look like." Thus, any code that uses a particular interface knows what methods might be called for that interface, and that's all. So interfaces are used to establish protocols between classes, they promote flexibility and reusability in code by connecting objects in terms of what they can do rather than how they do it.

A class **implement**s an interface by implementing all the methods contained in the interface. In contrast, inheritance by subclassing passes both a set of methods and their implementations from superclass to subclass. Whereas a class can inherit from only one superclass, a class can implement as many interfaces as it chooses to.

Notation

The Rumbaugh notation [3] is used to depict Java object models throughout

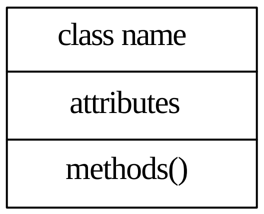

Figure 6: Rumbaugh Notation for depicting a Class

this thesis. In the Rumbaugh notation a class is depicted as shown in Figure 6. The top-most rectangle is used for the class name. The class attributes are represented in the middle rectangle and the bottom rectangle is used to show the methods that can

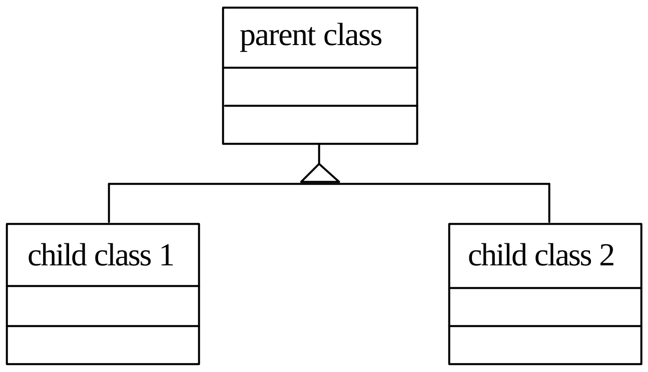

Figure 7: Rumbaugh Notation for depicting Inheritance

be called on or by the class. An inheritance relationship is depicted in the Rumbaugh notation as shown in Figure 7.

3.2.3 Remote Method Invocation

In distributed object systems, communication between program-level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require Remote Method Invocation or RMI [10]. In such systems, a local surrogate (stub) object manages the invocation on a *remote object*. The Java language's RMI system assumes the homogeneous environment of the JVM, and the system can therefore take advantage of the Java object model whenever possible.

In order to be a remote object, the definition of the corresponding class is required to implement a *remote interface*. A remote interface is one that extends the interface java.rmi.Remote which is defined in the Java API [9]. RMI treats a remote object differently from a local object when the object is passed from one JVM to another. Rather than sending a copy of the implementation object to the receiving JVM, RMI passes a remote stub for the remote object. The stub implements the same remote interface as the remote object and acts as its local representative, and basically is, to the caller, the remote reference.

RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates some remote objects, registers them in the RMI Registry (a standard Java tool included in Java Developer Kit, a.k.a. JDK), and waits for clients to invoke methods on these remote objects. A client application gets references to remote objects in the server from the RMI registry and then invokes methods on them.

Figure 8 depicts an RMI distributed application that uses the Java registry service to obtain a reference to a remote object. The server calls the registry to associate or bind a name with a remote object, which results in the creation of a corresponding stub object in the registry. The client looks up the remote object by its name in the server's registry, which causes the stub object to be transferred to the client and the stub class bytecode downloaded from the server over the HTTP

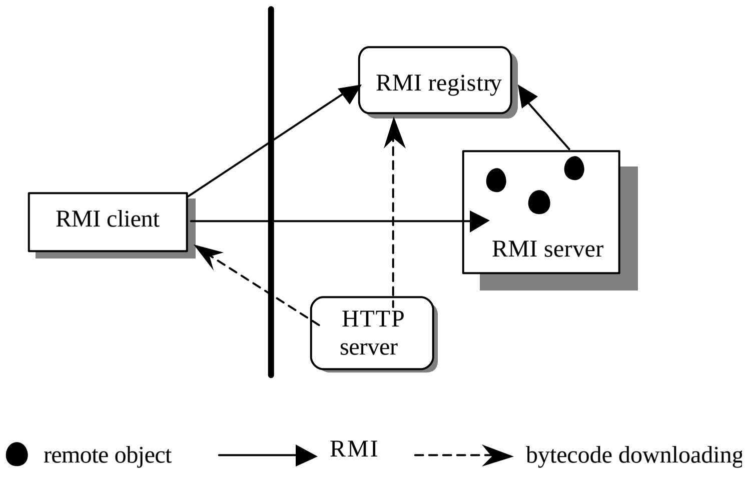

Figure 8: Remote Method Invocation

protocol. Now the client can invoke methods on the stub, which is responsible for carrying out the method call on the remote object.

3.2.4 Dynamic Class Loading

The Java language's portable and interpreted nature produces a highly dynamic and dynamically extensible system. While the Java compiler is strict in its compile-time static checking, the language and run-time system are dynamic in their linking stages: class bytecodes are loaded and linked only as needed, new code modules can be linked in on demand from a variety of sources, even from sources across a network, which enables dynamic and transparent updating of applications.

The *default class loader* is used by the Java runtime environment to load an application class (whose main method is run by using the java command) from local class directories specified by CLASSPATH, the environment variable that stores a list of directory names. All classes used directly in that class (i.e., classes needed to instantiate objects via the new expressions) are subsequently loaded by the default class loader from the local CLASSPATH whenever they are first referenced.

Defined by the Java class java.rmi.server.RMIClassLoader is the *RMI class loader* that provides a set of methods for the RMI system to download classes over the network, for instance, stub classes associated with remote objects and subclasses associated with objects passed as arguments and return values in RMI calls.

Java programmers can use the default class loader and the RMI class loader to mimic Java runtime's dynamic class loading behavior, which is exactly what we have done to implement the class loader in JEMS (see Section 3.3.3). For a deeper and better understanding of the rather complex mechanics of dynamic class loading and linking, readers are referred to [10].

3.3 The JEMS Architecture

The architecture of JEMS is depicted in Figure 9:

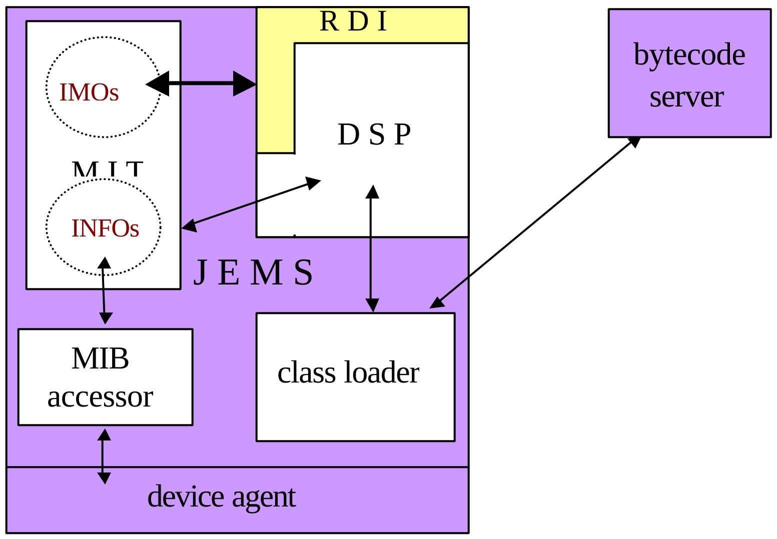

Figure 9: The JEMS Architecture

Management Information Tree (MIT). A "container" that holds Java objects in a tree structure. Two kinds of objects are stored in the MIT: (1) IMOs that are delegated by remote managers to the JEMS; they perform monitoring functions, i.e., observation or analysis, and interact with remote managers via the RDI. (2) INFormation Objects (INFOs) which, as their names suggest, store management information in an object-oriented way; they are used by IMOs to implement monitoring functions.

Delegation Service Provider (DSP). An RMI server object using the MIT and the class loader to implement the RDI; it provides the delegation service needed by remote managers to delegate, control and communicate with IMOs.

Class Loader. An internal Java object used by the DSP to load, either locally or from some class server (a.k.a. bytecode server) on the network, those classes that are needed to instantiate corresponding delegated objects.

MIB Accessor. An internal Java object used by INFOs to exchange low-level management information with the local MIB. In our work, we have implemented an SNMP accessor that talks with SNMP agents.

3.3.1 Management Information Tree

The management information tree (MIT) is a container object that JEMS

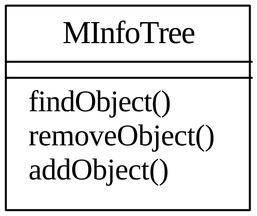

Figure 10: The MInfoTree Class

employs to store and manage objects in a tree structure. The MIT is instantiated from the MInfoTree class (Figure 10), which provides methods to add objects to, remove objects from and find objects in the MIT. Two kinds of objects are store in the MIT: INFOs and IMOs, which are respectively instantiated from subclasses of the MgmtInfo class and the Monitor class, both of which are derived from the superclass MO (Figure 11).

Object Naming Convention

Every object in the MIT has a name that uniquely identifies it and implies its location in the tree. The naming structure for INFOs is compatible with that of a

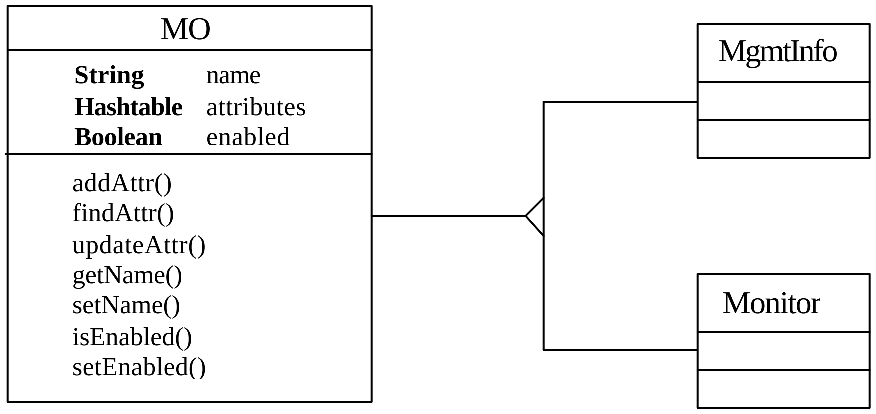

Figure 11: Inheritance Tree of Managed Objects

directory, and the hierarchy chosen is based on *containment*, i.e., an INFO is named in terms of the INFO representing the network resource that contains the resource that is represented by the INFO to be named. For example, if the INFO representing an ATM-switch port (say, with gloabal index = 1) has the name system:port.1, then the object modeling an incoming virtual path (say, with VPI = 2) in this port will have the name system:port.1:inPath.2; and accordingly in the MIT, this virtual path object will be stored as one of the child objects of the port object. The root of the MIT is a special dumb object named system.

Since IMOs don't represent network resources, there are no similar hierarchies for their organization, therefore we have made a decision to contain them directly under the MIT root. For instance, an IMO that computes a ratio can have such a name as system:ratio#3. The bottom line here is, any consistent scheme that guarantees naming uniqueness should be acceptable.

The MO Class

Representing network management information requires modeling those aspects that are of interest to network management purpose. The result of this abstraction in an object-oriented context is a managed object class (MO) consisting of a set of attributes and methods.

Managed objects use a hashtable attributes to store the list of properties that are of interest to management purpose. Each such property is represented by an

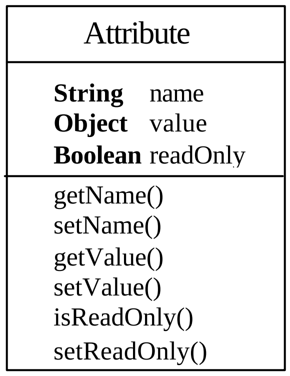

Figure 12: The Attribute Class

attribute object instantiated from the Attribute class (Figure 12). Any attribute object has a name and a value, and the name is used by the managed object as the key to index the attribute in the hashtable. The flag readOnly, if set to true, prevents the value of the attribute from being modified once initialized.

Each managed object has a unique name that specifies its identity and implies its location in the tree structure. This name is given when the object is initially created and remains immutable throughout its lifetime. “enabled” is a flag variable used to control and indicate whether the the managed object is activated or not.

MO has two very important subclasses: MgmtInfo and Monitor, which represent physical/logical network resources and network monitoring functions respectively.

Before looking into details of these two classes, we would like to point out that managed object classes are formally specified using the Guidelines for the Definition of Managed Objects (GDMO) [1], which is an object-oriented information specification language with management orientation. However, since GDMO definitions are text files that are difficult to read and interpret, we have taken advantage of Rumbaugh notation to visualize class specifications instead of bringing out GDMO files directly, and will continue doing so throughout the thesis. It is also relieving to know that there have been successful research efforts to translate GDMO definitions to Java objects [6], which provides automated tools to help with Java implementation of GDMO-specified MOs.

The MgmtInfo Class

All the INFOs are instantiated from proper subclasses of the MgmtInfo class, an object-oriented modeling of network management information. The values of interesting MIB variables are obtained through the MIB accessor and represented by corresponding attributes in a certain INFO. For example, for every incoming virtual path in any active port of a Fore ATM switch, the SNMP agent keeps track of the values of a collection of MIB counter variables such as pathUsedBandwidth, pathCells, pathUptime, and pathRejectedCells, just to name a few. Therefore, an INFO that models this virtual path will have in its attribute hashtable a list of attribute objects that correspond to these MIB variables. These attributes are added to the hashtable by invoking the addAttr method in the INFO's constructor function:

```
// inheritance from MgmtInfo
class InVirtualPath extends MgmtInfo {
        // constructor
        InVirtualPath (String name) {
                addAttr( new AttrInt("pathUsedBandwidth") );
                addAttr( new AttrInt("pathCells") );
                addAttr( new AttrInt("pathRejectedCells") );
                . . .
                // assign the name to this object
                this.name = name;
        }
        . . .
}
```

Therefore, creation of an incoming-virtual-path INFO looks like this:

```
InVirtualPath p2 = new InVirtualPath("system:port.1:inPath.2");
mit.addObject( p2.getName(), p2 );
```

The first Java statement instantiates a new INFO of the type InVirtualPath (p2) and gives it a unique name system:port.1:inPath.2. The second statement inserts p2 into the MIT at the desired location implied by its name.

All the INFOs of interest to network monitoring are instantiated and added to the MIT automatically when the JEMS starts up; they use the MIB accessor to periodically retrieve data from the underlying device agent, e.g., an SNMP agent, and use the acquired data to update the values of their attributes.

The Monitor Class

IMOs are instantiated from corresponding subclasses of the Monitor class. They are delegated to the JEMS by remote manager applications, and perform network monitoring functions by processing management information retrieved locally from relevant INFOs and/or other IMOs. Please refer to Chapter 4 for a dedicated discussion of the Monitor class and various IMOs instantiated from it.

3.3.2 Delegation Service Provider

The Delegation Service Provider (DSP) is an RMI server object instantiated from the DSProvider class, an implementation of the Java remote interface IfRDI which is actually the RDI specified in the Java language. The DSProvider class implements the IfRDI methods (or RDI operations) as follows:

void create(String className, String objName, Hashtable initParams)

The pseudo-code for this method looks like the following:

```
if (mit.findObject(objName) == null) {
        Class c = GenericLoader.loadClass(className);
        Monitor obj = (Monitor) ( c.newInstance() );
        obj.setName(objName);
        obj.initialize(initParams);
        mit.addObject(obj);
}
```

It first checks whether the IMO named objName already exists in the MIT; if not, it uses the class loader to load the bytecode of the associated class specified by the argument className, instantiates an object from this class, sets the object’s name to objName, further initializes the IMO, and then adds it into the MIT.

void delete(String objName)

This method removes the IMO objName from the MIT:

```
MO obj = mit.findObject(objName);
if (obj != null) { mit.removeObject(obj); }
```

void enable(String objName)

It enables the IMO objName by setting its flag variable enabled to true. By doing so, the object's functions are effectively turned on, since the communication operations, get and set, will first consult this variable to make sure that it is set to true before proceeding to accomplish their perspective tasks:

```
MO obj = mit.findObject(objName);
if (obj != null) { obj.setEnabled(true); }
```

void disable(String objName)

It disables the IMO objName by setting its flag variable enabled to false.

```
MO obj = mit.findObject(objName);
if (obj != null) { obj.setEnabled(false); }
```

Hashtable get(String objName, String namePattern)

This method finds, in the IMO named "objName," all the attributes whose names match the pattern specified in namePattern, and returns a hashtable of these attributes indexed by their names. The remote manager can then use the methods in the attributes themselves to get their values. Two wildcard characters, '*' and '?', could be used in the name pattern. '*' matches any number and combination of characters that can be part of a valid attribute name; while '?' can match any single character. For instance, 'path*', '*Cell*', 'path????Count' and 'pathCellCount' are all valid name patterns.

void set(String objName, Hashtable attrValues)

This method tries to set the values of some attributes in the IMO objName. The hashtable attrValues stores a list of values indexed by the names of the attributes which these values are meant for.

3.3.3 Class Loader

The class loader is a Java object instantiated from the GenericLoader class. It provides a static method loadClass to load class bytecodes, either locally or remotely from over the network, whenever they are needed for object(s) instantiation. By using this generic class loader, the locations of class bytecodes become transparent to the caller, i.e., the delegation service provider.

The GenericLoader Class

The GenericLoader class has only one public method loadClass, which accepts a String argument as a fully-qualifying class name and tries to return the bytecode for this class in a corresponding Class object. The DSP then uses this returned Class object to instantiate the desired IMO (see Section 3.3.2). This loadClass method first tries to load the class bytecode, if not already loaded in memory, from the local CLASSPATH using the default class loader:

```
Class c = Class.forName(className);
```

If the bytecode is not locally available, a further attempt is then made to download it from the network:

```
c = RMIClassLoader.loadClass(url, className);
```

url is a URL object that represents a Uniform Resource Locator pointing to a host, e.g., http://ux7.sp.cs.cmu.edu:2001/, where the needed bytecode is expected to be located. HTTP is the standard protocol for RMI to transfer bytecodes over the network, which means that there has to be an HTTP server running on the host ux7.sp.cs.cmu.edu. This server needs not be a full-scale heavy-weight Web server though, the ability to handle HTTP GET requests will suffice. Such a light-weight server can be downloaded free of charge from Sun's website.

Class Loader Configuration

The URL used by the class loader is obtained from a special INFO named system:classLoaderConfig, which has only one attribute, urls, that stores a list of URL addresses of bytecode servers. When the class loader needs to download bytecodes from the network, it retrieves this list and tries the URLs one at a time until one bytecode server responds with the desired Class object.

This INFO is special in the sense that it does not relate to or represent any management information or functionality per se; instead, it is a convenience object used by the class loader for configuration information storage. However, we have put it in the MIT so that the remote manager can take advantage of the set operation in the DSP service to change the value of urls on the fly, as per possible configuration changes of bytecode servers in the network.

This configuration object is created automatically during the JEMS startup, with urls initialized to a default value.

3.3.4 MIB Accessor

Similar to the class loader, the MIB accessor is also an internal object invisible to remote managers. It is used by INFOs in the MIT to exchange management data with the underlying device agent. Currently we have only implemented an SNMP accessor that talks with SNMP agents. However, accessors that communicate with other types of agents, e.g., CMIP agents or vendors-specific agents, can be built in a similar way.

The SNMPAccessor Class

The SNMP accessor is a Java object instantiated from the SNMPAccessor class (Figure 13). It is a very small SNMP manager that communicates with the local

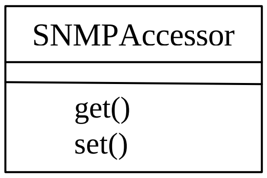

Figure 13: The SNMPAccessor Class

SNMP agent in the network device where the JEMS is running. We have used the most popular Java SNMP API [11] to implement the accessor.

When the accessor is first created during the JEMS startup, it loads the correct MIB file, sets the SNMP protocol version (v1, v2c or v3) and the Internet address for the SNMP agent (localhost in our case). The accessor has some private methods that help parse variable names, translating them to corresponding Object Identifiers (OIDs) which are eventually encoded in SNMP protocol data units. The accessor exposes two public methods, set and get, for INFOs to invoke, hiding all the agent-related details from them.

Chapter 4 Intelligent Monitoring Objects

The JEMS architecture provides for a stage on which intelligent monitoring objects (IMOs) can perform. Certain network monitoring functions, which used to be running on the manager side, can now be encapsulated into corresponding IMOs, which are in turn delegated to the JEMS and running locally in the managed network device. This chapter focuses on the details of the design and implementation of IMOs.

From time to time, however, we will refer back to certain JEMS components and have further discussions about them, since we either simplified their presentations or didn't make their design considerations clear enough in the last chapter. We have purposefully done this because their thorough interpretation relies on or is made easier by the contents of this chapter.

Chapter Organization

Section 4.1 analyzes the requirements on IMO design.

Section 4.2 details the design of the Monitor class.

Section 4.3 examines the implementation of *Observer*s, i.e., IMOs that compute rate/ratio-oriented indicators and indicator statistics (review Section 2.4.1).

Section 4.4 examines the implementation of *Analyzer*s, i.e., IMOs that perform network analysis (review Section 2.4.2).

Section 4.5 illustrates, via a simple example, how to achieve intelligent network monitoring by delegating proper IMOs to the JEMS.

4.1 Design Analysis

As was introduced in Chapter one and three, IMOs are specialized Java objects that are delegated by manager applications to a JEMS to perform network monitoring right in the managed network device, and their functions allow for on-line modification and extension. In order to satisfy their functional requirements, IMOs need to have the following characteristics:

(1) They have access to lower-level management information, i.e., attributes values stored in related INFOs or even other IMOs.

(2) A self-contained monitoring module is properly encapsulated in every IMO; once correctly initialized, it is able to autonomously exercise a desired monitoring function/computation without any manager intervention.

(3) Through relevant RDI calls, remote managers can make on-line changes or extensions to the functionality of IMOs.

(4) Although IMOs are instantiated and run in the network devices, they have to be coded at the manager side with no access to agent-side management information. This requires that the way in which IMOs are designed support easy manager-side object programming.

We have also noticed that the monitoring modules in different IMOs are highly "repetitive." For example, the formula used for different ratio calculations are mathematically equivalent except that different ratios are based on different lower-level management information. Therefore, there is one last consideration:

(5) Maximum reusability of monitoring functions in different object instances

is expected in the IMO design.

4.2 The Monitor Class

All monitoring objects are instantiated from proper subclasses of the abstract Monitor class which forms the core of the whole IMO design. The details of the

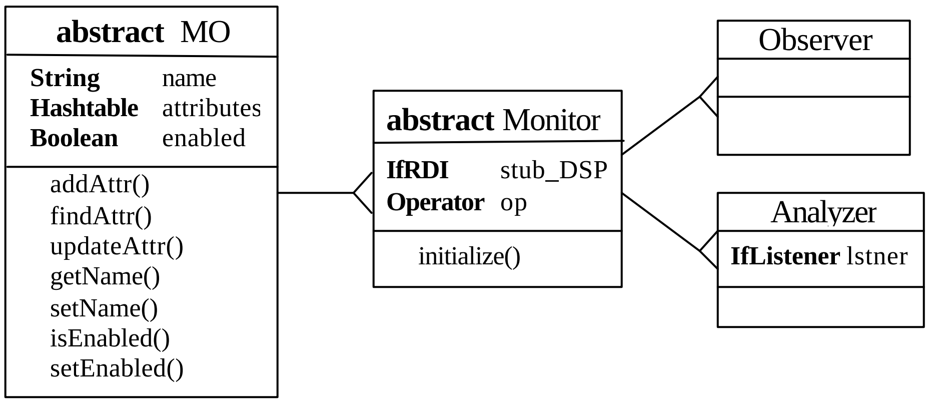

Figure 14: Inheritance Tree of the Monitor Class

class structure are discussed as follows: Section 4.2.1 shows how requirements (1) and (2) are satisfied through the so-called “operands/operator paradigm”, Section 4.2.2 examines the initialization process of IMOs, and Section 4.2.3 presents the benefits of our design, where requirements (3), (4) and (5) are addressed

4.2.1 Operands/Operator Paradigm

It can be noticed that all the IMOs, no matter they are Observers or Analyzers, accomplish their functionality by taking given management information as *input*, applying certain monitoring *rule*s to it, and generating some *result*s. For an Observer, the result is stored in a certain attribute ready for retrieval by the remote manager; for an Analyzer, the result, usually in the form of an alarm, will be reported back to interested AlarmListener objects on the manager side.

Accordingly, we have generalized the IMO design using a structure which we call Operands-Operator Paradigm (Figure 15). It consists of three essential components, the operands attribute, the operator attribute with the corresponding Operator object, and the result/listener attribute, which correspond to the above-mentioned *input*, *rule* and *result* respectively. We now examine these components separately.

The operands Attribute

In the attribute list (i.e., Hashtable attributes) of every IMO, there is a

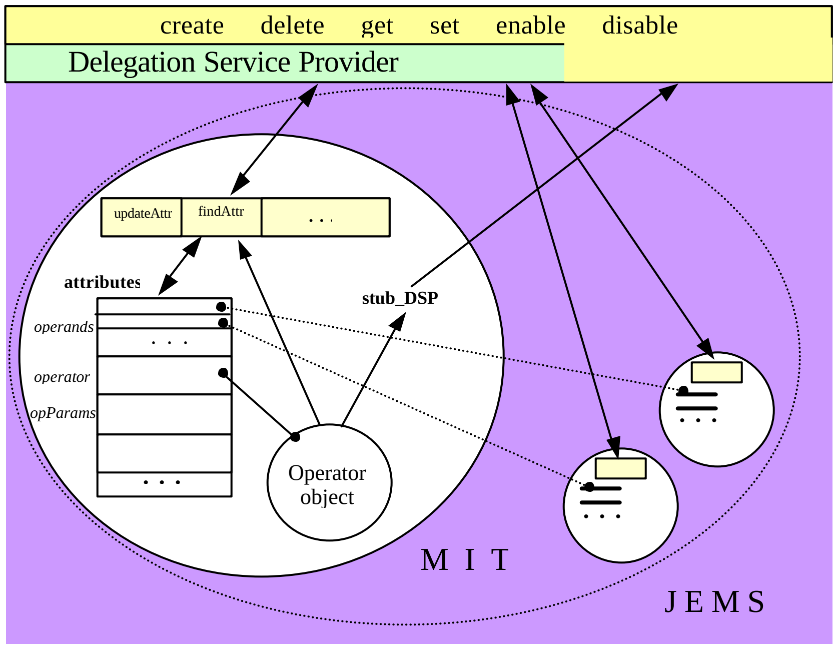

Figure 15: Operands-Operator Paradigm

composite attribute named operands, which represents a String array of absolute attribute names. The syntax of an absolute attribute name is

<attributeName>@<objectName>

and such a name refers to a given attribute in a given managed object. For example:

pathCells@system:port.1:inPath.2

The names stored in the operands attribute can point to either attributes in INFOs or the result attribute in other IMOs, depending on the operation to be performed by the IMO in question. The Operator object in a IMO retrieves the values of those attributes specified by the operands, performs pre-programmed computation on the values, and stores the output into the result attribute of the IMO in question (in the case of an Observer). The operands attribute is a read-only attribute; once a IMO is initialized, the value of this attribute cannot be changed by remote managers.

The operator Attribute and Operator Object

The monitoring behavior of IMOs are generalized by the abstract Operator class (Figure 16). Any specific monitoring functionality, e.g., ratio or rate

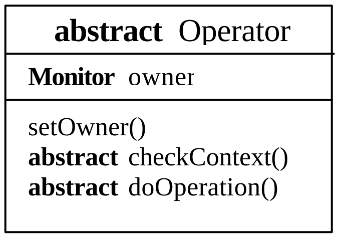

Figure 16: The **abstract** Operator Class

computation, is represented by a subclass that has accordingly implemented Operator's abstract methods. The purpose of each method is explained as follows:

setOwner() – Set the owner of this Operator to be the associated IMO

checkContext() – Check if the owning IMO has the suitable context for the Operator to run. Such a context includes compatible operands (the operands attribute), proper operation parameters (the opParams attribute) and availability of a listener stub object (in the case of an Analyzer).

doOperation() – Perform the desired monitoring operation based on the valid operands and operation parameters; specific to operator implementation.

Any IMO has, in its attribute list, an attribute named operator whose value is the class name of the Operator to be used by the IMO. This attribute is not read-only, and whenever it is set to a new value, a corresponding Operator object will be instantiated using the generic class loader.

Also related to the Operator object is the attribute named opParams. It represents a hashtable of parameters, in name/value pairs, that are interpreted by the Operator to adjust its operation. The usage of opParams will be discussed in more detail later in this chapter where most appropriate.

The result/listener Attribute

For an Observer, the output from the Operator object is stored in an attribute named result. This attribute can be retrieved by remote managers, or even taken as an operand by other IMOs. An Analyzer, instead, has a listener attribute whose value is the name of a remote listener object; the Operator object invokes a method on the stub of this remote listener to send an alarm back to the manager whenever an abnormal condition is detected. A better study of these two attributes will be given in Section 4.3 and 4.4 repectively.

4.2.2 Object Initialization

The correct and efficient initialization of IMOs is an integral part of the overall design, therefore we are devoting a separate subsection to this topic. There are two steps involved here: invocation of the object’s constructor and initialize method by the create operation of the DSP (review Section 3.3.2).

The Constructor Method

A IMO’s constructor is automatically invoked immediately after the object is instantiated by the DSP. What the constructor does is to correctly set up the object’s attribute list by adding appropriate attributes into the hashtable, and to initialize its stub_DSP property (see Figure 14, 15). To make this process clearer, we list in the following paragraphs partial codes of the related constructor methods, with easy-to-understand comments.

```
// the abstract Monitor class
abstract class Monitor extends MO {
        // constructor
        Monitor() {
                // find the DSP on the localhost and store its stub
                // object into the variable “stub_DSP”
                stub_DSP = (IfRDI) Naming.lookup(“//localhost/DSP”);
                // add the read-only “operands” attribute
                addAttr(new Attribute(“operands”));
                // add the read-write “operator” attribute
                addAttr(new Attribute(“operator”, false));
                // add the read-write “opParams” attribute
                addAttr(new Attribute(“opParams”, false));
                // add the read-write “interval” attribute
                // refer to Section 4.3 and 4.4 for details
                addAttr(new Attribute(“interval”, false));
        }
        . . .
}

// the Observer class
class Observer extends Monitor {
        // constructor
        Observer() {
        // call the constructor of the parent class – Monitor
                super();
                // add the read-only “result” attribute
                addAttr(new Attribute(“result”));
        }
        . . .
}

// the Analyzer class
class Analyzer extends Monitor {
        // constructor
        Analyzer() {
                // call the parent class’ constructor
                super();
                // add the read-write “listener” attribute
                addAttr(new Attribute(“listener”, false));
        }
        . . .
}
```

The initialize Method

An IMO’s initialize method is called immediately after the constructor, and it aims at correctly initializing the values of the those attributes added to the attribute list by the constructor.

```
// the abstract Monitor class
abstract class Monitor extends MO {
        // constructor
        Monitor() { . . . }
        void initialize(Hashtable params) {
                // initialize “operands” with the corresponding value in
                // the hashtable “params”, which is passed as the last
                // argument to the DSP’s create() method (Section 3.3.2)
                updateAttr( “operands”, params.get(“operands”) );
                updateAttr( “opParams”, params.get(“opParams”) );
                updateAttr( “interval”, params.get(“interval”) );
                updateAttr( “operator”, params.get(“operator”) );
        }
        . . .
}

// the Observer class
class Observer externds Monitor {
        // constructor
        Observer() { . . . }
        void initialize(Hashtable params) {
                // call Monitor's initialize method
                super.initialize(params);
        }
        . . .
}

// the Analyzer class
class Analyzer externds Monitor {
        // constructor
        Analyzer() { . . . }
        void initialize(Hashtable params) {
                // call Monitor's initialize method
                super.initialize(params);
                // initialize the "listener" attribute
                updateAttr("listener", params.get("listener"));
        }
        . . .
}
```

The updateAttr method acts as a dispatcher function that redirects the initialization process to respective "helper" methods. The attribute value to be set, which is the second argument to updateAttr, is passed on to the helper methods. For instance, the call to initialize the operands attribute:

```
updateAttr( "operands", params.get("operands") );
```

results in the following helper method to be called:

```
private void updateOperands(Object value) { . . . }
```

These helper methods perform type checking, value assignment and additional setup if necessary. Take the operator attribute as an example, the updateOperator method first checks whether the value to be set is a String instance. If so, it assigns the value to the operator attribute. That's not all, there is some additional setup work to be done in this case: first, the generic class loader loads the associated operator class and instantiates an Operator object; secondly, a Java thread is created and started, which calls the Operator's doOperation method periodically at the pace specified by the interval attribute (for details, please see Section 4.3):

```
// Monitor's updateOperator method
private void updateOperator(Object value) {
        // assign the value to "operator"
        findAttr("operator").setValue((String)value);
        // load the operator class
        Class c = GenericLoader.loadClass((String)value);
        // instantiate the Operator object
        op = (Operator) (c.newInstance());
        // set the Operator's owner to be this IMO
        op.setOwner(this);
        // check the context of this IMO
        op.checkContext();
        if (findAttr("interval").getValue() > 0) {
                // create and start a thread that calls
                // op.doOperation() periodically
        }
}
```

4.2.3 Benefits of Our Design

The operands/operator paradigm has the following benefits.

Operator Reusability

A specific monitoring function can be abstracted and represented by a proper subclass of the Operator class, and then be embedded into different IMOs in the form of the operator attribute. In this way, we can bind one operator with many different sets of operands to form different IMO instances.

On-line Change and Extension

Flexible and powerful on-line extension to IMOs and the underlying JEMS can be achieved. Specifically speaking, there are three levels of on-line changes that can be made, with increasing degree of extensibility.

(1) Parameter Customization. Because the opParams is a read-write attribute, remote managers can modify its value via RDI calls, thus dynamically changing the manner in which the Operator object works.

(2) Operator Re-assignment. Since the Operator itself is associated to the read-write operator attribute, managers can change a IMO's monitoring behavior even more by assigning a new, hopefully more powerful, Operator to it. Note that the Operator object is instantiated by way of the generic class loader which can download Java classes from the bytecode server, the JEMS does not need to have any prior knowledge or storage of the new operator class. Therefore, the extension takes place in a completely on-line and dynamic fashion.

(3) New Monitor Types. The most significant benefit comes from both the operands/operator paradigm and the overall JEMS design, i.e., it is quite easy and convenient to write new Monitor subclasses and to delegate their instances to the JEMS.

Independent Manager-side Development

To write and compile the Monitor class and its subclasses, we need:

Interfaces/classes: IfRDI, IfListener, Operator, Attribute

To write and compile the Operator class and its subclasses, we need:

Interfaces/classes: IfRDI, IfListener, Monitor, Attribute

To deploy/delegate IMOs to the JEMS, we also need to have access to related

Operator implementations, IfListener implementations, and absolute attribute names from available INFOs.

Except for IfRDI and absolute attribute names, all of the above-mentioned components are written and available at the manager side. IfRDI is the remote interface to the JEMS and, according to the purpose and nature of an interface, should remain independent of its implementation. As for the absolute attribute names, as long as the managed device is determined and a mapping from its MIB to corresponding INFOs is done, the available attribute names are fixed and become irrelevant to the JEMS implementation. Therefore, our design allows for independent manager-side IMO programming and deployment.

4.3 Observers

Observers are IMOs instantiated from the Observer class. They perform the observation part of network monitoring. An Observer is characterized by the

Observer type	Operator class	Context			
		interval (ms)	operands number	operands type	opParams
rate	OpRate	≥ 1	= 1	Integer or Float	n/a
ratio	OpRatio	≥ 1	≥ 2	Integer or Float	n/a
average	OpAvg	≥ 1	= 1	Integer or Float	sampleSize ≥ 2
variance	OpVar		= 1		
covariance	OpCovar		= 2		

Table 3: Observer Contexts for different Operators

Operator object in it, therefore we will center our discussion around related Operator objects. The following table lists the correct context for different Operators:

Let's take a look at how a ratio-oriented indicator is implemented by creating an Observer using the OpRatio class. Other indicators can be implemented similarly. The manager creates an initialization parameter list:

initParams

operands	"pathRejectedCells@system:port.1:inPath.2" "pathCells@system:port.1:inPath.2"
opParams	(null)
Interval	5000
operator	"jems.operator.OpRatio"

Table 4: Initialization Parameters for the OpRatio Class

The manager delegates an Observer to the JEMS (at rrocoto.cshcn.umd.edu):

```
IfRDI jems = (IfRDI)Naming.lookup("//rrocoto.cshcn.umd.edu/DSP");
jems.create("jems.monitor.Observer", "system:observer#58", initParams);
```

The DSP instantiates an Observer, initializes its attribute list using the provided parameters, and adds the Observer into the MIT. When the operator attribute is being initialized, the Operator checks the owning Observer against the ratio context listed in Table 3 to make sure that the Observer has a compatible context for the Operator to correctly perform its function in.

The monitoring funtion of this Observer, i.e., to calculate the (cell rejection) ratio based on the two variables specified in the operands attribute, is implemented by OpRatio's doOperation() method as follows:

```
// OpRate's doOperation method
void doOperation() {
        // get the value of the "operands" by calling:
        // owner.findAttr("operands");
        // parse the names of the operands and get the attribute
        // names and object names separately, i.e.:
        // attrName1 = "pathRejectionCells"
        // objName1 = "system:port.1:inPath.2"
        // attrName2 = "pathCells"
        // objName2 = "system:port.1:inPath.2"
        // get the attribute values via two RDI calls:
        // curr_1 = (owner.stub_DSP).get(objName1, attrName1);
        // curr_2 = (owner.stub_DSP).get(objName2, attrName2);
        // calculate the difference from the latest values:
        // delta_1 = curr_1 – last_1;
        // delta_2 = curr_2 – last_2;
        // calculate the ratio by:
        // ratio = delta_1 / (delta_1 + delta_2);
        // store the ratio to the “result” attribute:
        // owner.findAttr(“result”).setValue(ratio);
        // last_1 = curr_1;
        // last_2 = curr_2;
}
```

4.4 Analyzers

Analyzers are IMOs instantiated from the Analyzer class. They perform the analysis part of network monitoring as was defined in Section 2.3.2. In this thesis work, we only look into the simplest form of Analyzer, which compares the value of the monitored indicator (its operand) with a set of pre-specified thresholds, and generates corresponding alarms when certain thresholds are crossed. The comparison is done by the Analyzer’s Operator object which is instantiated from the OpThreshold class. Other types of operator classes can be developed similarly.

Figure 17 shows the threshold comparing model used by an Analyzer. There

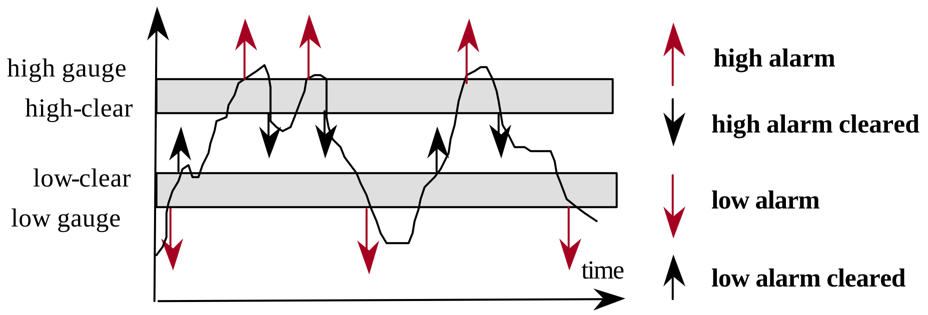

Figure 17: Threshold Comparing in Alarm Generation

are two levels of thresholds: high and low, and in each level there are two gauges defined. For the high-level thresholds, when the value of the monitored indicator exceeds the high gauge, a "high alarm" is fired and remains valid until the value drops below the high-clear gauge, when a "high alarm cleared" message will be emitted. Between the firing of an alarm and its clearance message, no other alarms should be fired. The "low alarm" and its associated "low alarm cleared" message work in a similar way. An even more generic model would allow for multiple levels of thresholds.

Alarms and their clearance messages are represented by objects instantiated from the Alarm and AlarmCleared classes respectively. Both these classes are

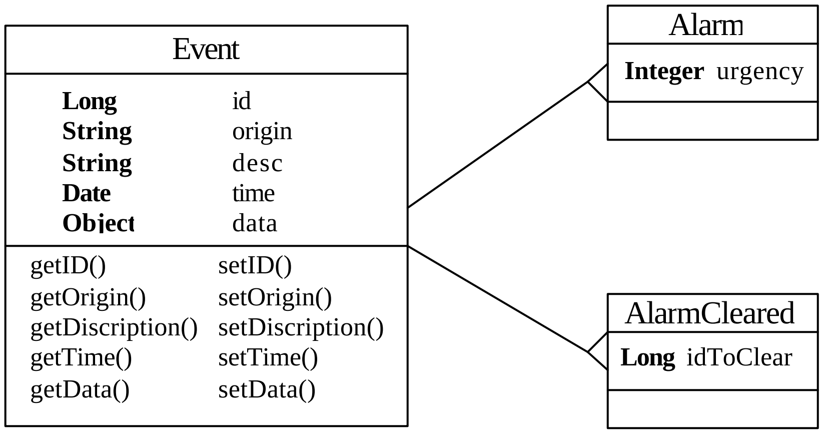

Figure 18: The Inheritance Tree of Event Classes

subclasses of the Event class (Figure 18). Each generated event object is automatically assigned a unique ID, and contains properties that have the following meaning:

origin	name of the IMO that generates this event
desc	brief description of the nature of this event
time	time stamp marking when the event was created
data	detailed information about the agent-side situations that triggered the

generation of this event; meant to be interpreted only by proper event handlers on the manager side

The generated event objects are sent back to the manager via a remote interface IfListener (Figure 19). A remote object running at the manager side has implemented this interface, and registered the associated stub object in the RMI registry under the name “EventListener”. When delegating an Analyzer to the JEMS, a manager includes the complete URL name of this stub object in the initialization parameter list (Table 5) for initialization of the listener attribute.

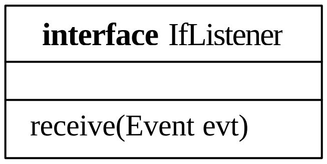

Figure 19: The IfListenter Remote Interface

Let’s now take a look at how threshold checking, a simple network analysis behavior, is instrumented by creating an Analyzer using the OpThreshold class. The manager creates an initialization parameter list for the OpThreshold class:

initParams

operands	“result@system:observer#58”
opParams	high_gauge = 80000 high_clear_gauge = 75000 low_gauge = 500 low_clear_gauge = 5000
interval	10000
listener	“//hera.isr.umd.edu/EventListener”
operator	“jems.operator.OpThreshold”

Table 5: Initialization Parameters for the OpThreshold Class

The manager delegates an Analyzer to the JEMS (at rrocoto.cshcn.umd.edu):

```
IfRDI jems = (IfRDI)Naming.lookup(“//rrocoto.cshcn.umd.edu/DSP”);
jems.create(“jems.monitor.Analyzer”, “system:analyzer#16”, initParams);
```

The DSP instantiates an Analyzer, initializes its attribute list using the provided parameters, and adds the Analyzer into the MIT. When the DSP initializes the listener attribute, in addition to the actual assignment of the URL to the attribute, the helper method updateListener also downloads a copy of the associated remote stub object and keeps it in the Analyzer's lstner property.

```
// Analyzer's updateListener method
private void updateListener(Object value) {
        // assign the value to "listener"
        findAttr("listener").setValue(value);
        // locates the remote stub of the "EventListener"
        lstner = (IfListener)Naming.lookup((String)value);
}
```

When the operator attribute is being initialized, the Operator checks the owning Analyzer against the context shown in Table 6 to make sure that the Analyzer has a compatible context for the Operator to correctly perform its function:

Analyzer Type	Operator class	Context				
		interval	operands#	type	opParams	listener
Threshold Checking	OpThreshold	≥ 1	= 1	Integer or Float	high_gauge > high_clear > low_clear > low_gauge	(not null)

Table 6: Analyzer Context for the OpThreshold Class

The monitoring funtion of this Analyzer is accomplished by the Operator's doOperation method as follows:

```
// OpThreshold's doOperation() method
void doOperation() {
        // get the value of the "operands" by calling:
        // owner.findAttr("operands");
        // parse the name of the operand and get the attribute name
        // and object name separately, i.e.:
        // attrName = "result"
        // objName = "system:observer#58"
```

```
            // get the value of the attribute via an RDI call:
            // value = (owner.stub_DSP).get(objName, attrName);
            // get the value of the “opParams” and the gauges
            // contained in it:
            // owner.findAttr(“opParams”); . . .
            // use the threshold model shown in Figure 4.12 and
            // compare “value” with different gauges to decide if
            // an Alarm or AlarmCleared object should be created;
            // if so, create one and initialize it accordingly
            // alarm = new Alarm();
            // alarm.setID(); alarm.setOrigin(owner.getName());
            // alarm.setDescription(“…”); . . .
            // send the alarm back to the manager:
            // lstner.receive(alarm);
    }
```

The remote listener object acts as an event dispatcher at the manager side and re-directs the received events to their proper handlers.

Chapter 5 Prototype JEMS System

Based on the JEMS architecture proposed in previous chapters, we have built a proof-of-concept prototype system, and done some experiment and comparison against the centralized monitoring system. The results have verified the advantages of intelligent monitoring over traditional network monitoring schemes.

Chapter Organization

Section 5.1 introduces the configuration and implementation of the prototype system.

Section 5.2 compares the prototype system with the traditional central monitoring system through some simple experiment and analysis.

Section 5.3 draws approriate conclusions.

5.1 Prototype JEMS System

The prototype JEMS system (Figure 20) is introduced in the following sections: system configuration in Section 5.1.1, objects deployment in Section 5.1.2,

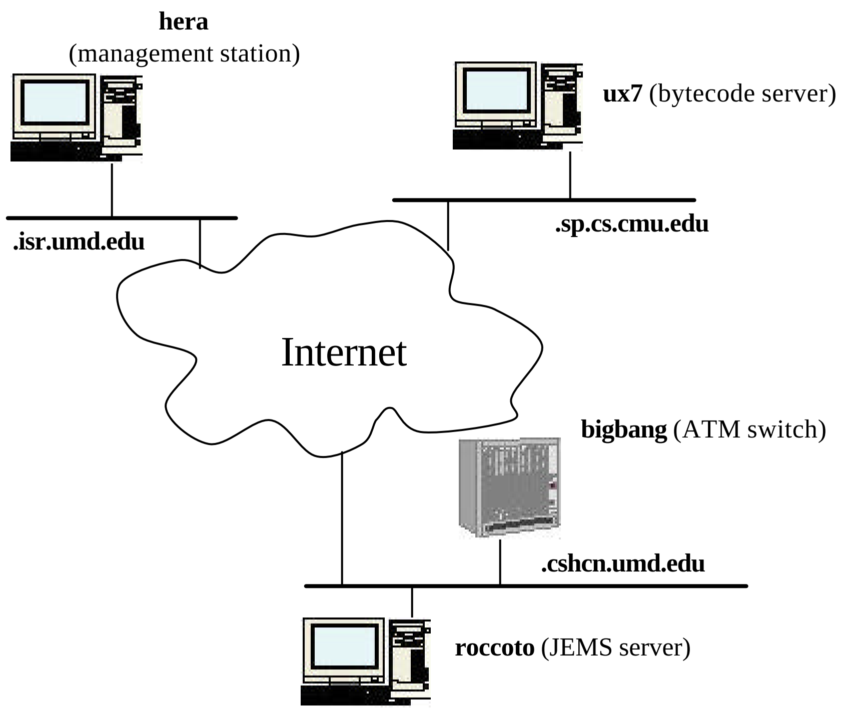

Figure 20: Prototype JEMS System

and system startup procedure in Section 5.1.3.

5.1.1 System Configuration

Bytecode Server and RMI Registry Service

Actually, the bytecode server and the RMI registry service can be running on any IP host on the Internet. We happen to choose the following setup, shown in Table 7, for them.

Machine's DNS name	ux7.sp.cs.cmu.edu
Hardware platform	Sun Workstation
Operating System	Solaris 2.5
Java platform	JDK 1.1.5 for Solaris
URL of RMI registry	rmi://ux7.sp.cs.cmu.edu:5001/
URL of HTTP server	http://ux7.sp.cs.cmu.edu:2001/
Classes served	jems.monitor.* jems.operator.*

Table 7: Configuration of the Bytecode Server

jems.monitor is the name of the Java package that contains all the Monitor classes, i.e., Monitor, Observer and Analyzer. jems.operator is the name of the Java package that contains all the Operator classes, i.e., OpRate, OpRatio, OpAvg, OpVar, OpCovar and OpThreshold.

Managed Device and JEMS Server

The managed device is a Fore ATM switch located in the Center for Satellite and Hybrid Communication Networks (CSHCN). The switch is connected to the CSHCN LAN and has the DNS name bigbang.cshcn.umd.edu. It has a built-in SNMP agent that serves requests for variables defined in two MIB files: RFC1213 for IP management, and Fore-Switch-MIB for ATM-switch-specific management.

According to the proposed architecture, a JEMS server should be running in the switch for its intelligent monitoring. However, since there is no JVM ported to the Fore ATM switch yet, we have to run a JEMS server in a workstation which is equipped with a JVM and acts as a *proxy* for the ATM switch. The configuration of this proxy machine is listed below in Table 8.

Machine’s DNS name	rrocoto.cshcn.umd.edu
Hardware platform	HP Workstation
Operating System	HP UX
Java platform	JDK 1.1.6 for Solaris

Table 8: Configuration of the Proxy Machine

The configuration of the JEMS server in the proxy machine is listed below:

MIT naming convention	as specified in Section 3.3.1
Name of DSP’s stub object registered in the RMI registry	*DSP* (i.e., remote managers use the URL *rmi://ux7.sp.cs.cmu.edu:5001/DSP* to look up DSP’s stub object)
SNMP agent accessed by the SNMP Accessor	bigbang.cshcn.umd.edu

Table 9: Configuration of the JEMS Server

Management Station

Any IP host on the Internet can play the role of a management station. We have randomly picked a Sun workstation in ISR’s SEIL lab with the following setup:

Machine’s DNS name	hera.isr.umd.edu
Hardware platform	Sun Workstation
Operating System	Solaris 2.5
Java platform	JDK 1.1.6 for Solaris
Name of event listener’s stub object registered in the RMI registry	*EventListener* (i.e., Analyzers in the JEMS use the URL *rmi://ux7.sp.cs.cmu.edu:5001/EventListener* to look up the event listener’s stub)

Table 10: Configuration of the Management Station

5.1.2 Managed Objects Deployment

Information Objects

Since the major interest of this research is in performance monitoring, we did

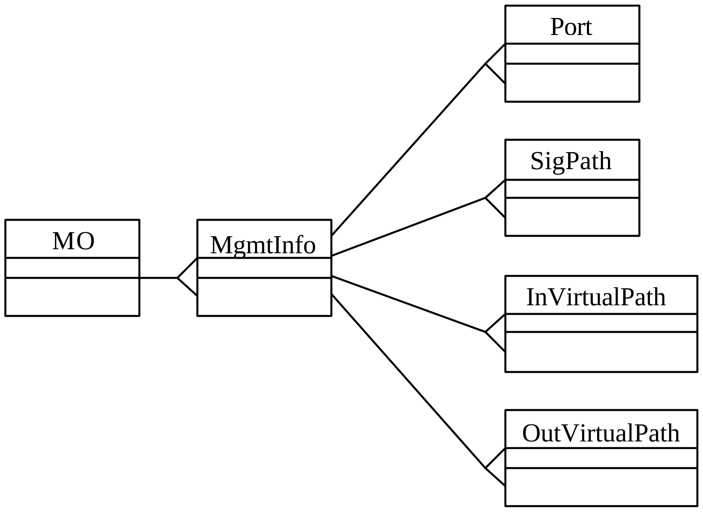

Figure 21: Inheritance Tree of INFO Classes

not map the whole Fore-Switch-MIB to INFOs. Instead, we have selected a list of SNMP variables related to the performance of the ATM switch (see Appendices), and mapped them to various INFOs. Specifically, all port-related variables (Table 11) are mapped to corresponding attributes in the Port class, all signaling-related variables (Table 12) are mapped to corresponding attributes in the SigPath class, all incoming path-related variables (Table 13) are mapped to corresponding attributes in the InVirtualPath class, and all outgoing path-related variables (Table 14) are mapped to corresponding attributes in the OutVirtualPath class. The inheritance tree of these classes is shown in Figure 21, and their associated containment subtree in the MIT is shown in Figure 22.

Now, the motivation and benefits of introducing a layer of INFOs between IMOs and underlying device agents are easy to explain and understand.

INFOs present a better view of the available management information. Not all the variables in the MIB are useful for network management, actually only a small fraction of them prove to be helpful for performance monitoring. To make

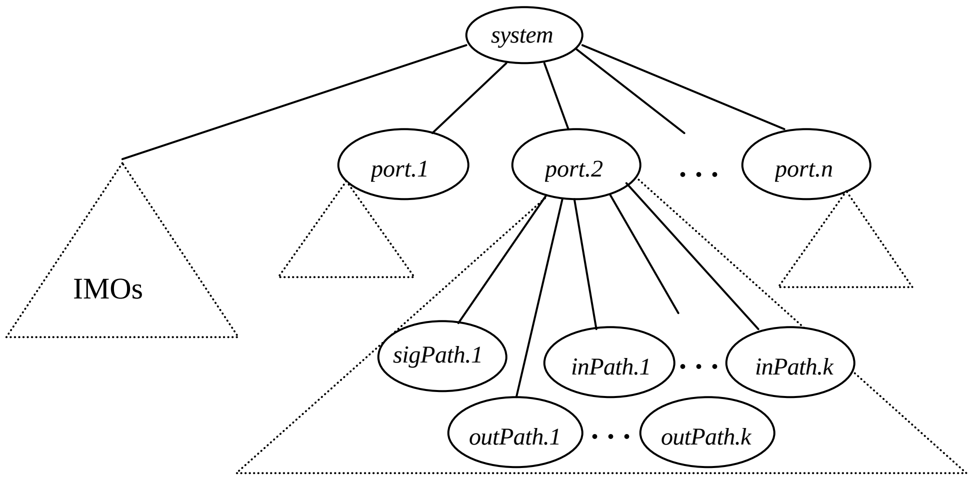

Figure 22: Containment Subtree of INFOs in the MIT

things worse, these variables are often scattered in different MIB tables, which makes their direct access very difficult and confusing. By mapping only those useful and functionally related variables from different locations in the MIB to attributes in a few compact INFOs, we hide the complexity of direct MIB access from end users (e.g. IMOs), and instead provide them with an efficient, focused and simpler access interface.

INFOs help with easy programming of reusable IMOs. Instead of talking directly to the device agent, IMOs acquire information from various INFOs by invoking proper RDI operations. Since MIB access details and complexity are encapsulated in the implementation of INFOs and the MIB accessor, IMOs are independent of agent types and are therefore reusable. For example, if we have an ATM switch with a CMIP agent, all that needs to be done is to write a CMIP accessor and change the implementation of INFOs so that they update their attribute values by way of the CMIP accessor.

Monitoring Objects

Two Java packages, namely, jems.monitor.* and jems.operator.*, are deployed to the bytecode server. When the remote manager attempts to delegate a IMO to the JEMS server, it invokes the RDI create method with three arguments: class name, object name and initialization parameter list. The Monitor class specified by the class name is downloaded from the bytecode server to the JEMS server, a IMO is then instantiated from the class, and gets initialized using the parameters provided.

5.1.3. System Startup Procedure

The prototype system starts up in the following sequential steps:

Bytecode Server and RMI Registry Startup

The HTTP server is started on port 2001, with its document root pointing to where the two Java packages are installed. Monitor and Operator classes are now ready for download from the URL http://ux7.sp.cs.cmu.edu:2001/.

The RMI registry is started on port 5001.

JEMS Server Startup

The SNMP Accessor is created and initialized. It loads the Fore-Switch-MIB file, sets the SNMP protocol version to v1, and sets the SNMP agent’s Internet address to bigbang.cshcn.umd.edu.

The Management Information Tree is created and initialized. First, the subtree shown in Figure 22 is automatically added into it; during this process, switch information is obtained via the SNMP Accessor to decide which ports are active and what virtual paths there are in each active ports, so that the subtree can be built to reflect the working condition of the ATM switch. Secondly, special-purpose objects are initialized and added into the MIT; for example, the classLoaderConfig object (see Section 3.3.3) will be added to the MIT with its attribute urls set to http://ux7.sp.cs.cmu.edu:2001/.

The Class Loader is created and initialized.

The Delegation Service Provider is created; its stub object is registered in the RMI registry under the name DSP, and therefore becomes available for lookup and download at the URL rmi://ux7.sp.cs.cmu.edu:5001/DSP.

Management Station Startup

A manager application starts up and immediately registers the stub object of an event listener to the RMI registry under the name EventListener. Now the startup process of the whole system is complete, and the manager application can start delegating Observers, Analyzers and possibly other types of IMOs to the JEMS server.

5.2 JEMS vs. Centralized Monitoring

This section compares the intelligent monitoring paradigm of JEMS and those of centralized SNMP-based approaches. It examines some typical performance issues involved in network monitoring systems. We compare the performance of our prototype system and applications using SNMP. The comparisons, which focus on scalability, performance, and online extensibility, are illustrated by simple examples or analysis.

5.2.1 Scalability

Polling-based network management systems do not scale up to large networks, because the interaction of the central management station with SNMP agents has two patterns: (1) it involves the management station into a huge amount of communication, and (2) it concentrates most processing into the central station. A network system becomes unmanageable when there is an increase in the number of managed devices or when there is an increase in the number of variables to be monitored.

Consider an SNMP-based application executing on hera, responsible for performance monitoring of the ATM switch bigbang. To undertake any non-trivial monitoring task, be it observation or analysis, the application needs to keep an array of indicators – rates, ratios and statistics – based upon the variables listed in Appendix. To provide a given level of measurement precision, the values of these indicators have to be updated at a reasonable frequency, say, every T seconds. If we assume the time required for a single polling request is t_P, then the maximum number of variables that can be handled by hera is bound by $N \leq T / t_P$. Monitoring an operating broadband ATM switch usually requires a high precision or a short T; and if the management station and managed device communicate over a Wide Area

Network (implying a large delay t_P), then the maximum number of variables or SNMP devices that can be handled by one central management station could drop one to two orders of magnitude. This poses a serious scalability problem. A smiliar example is discussed in much more detail in [26].

Now, even if the management station is so powerful as to be able to handle all the devices, the polling scheme is still very inefficient, since the network traffic caused by the polling behavior is a constant independent of the actual frequency of information access or alarm generation. For example, in order to promptly detect a rare yet important alarm situation, a high polling frequency needs to be maintained, even though most of the polling requests will prove to be irrelevant.

Contrary to what a traditional polling-based system does, a JEMS-based system delegates intelligent monitoring objects to network devices, so that various indicators are maintained (and network analysis is performed) right in the devices or in a local proxy as in our prototype system. Manager-to-agent traffic occurs only when the central station has the actual need for information, or when alarms are fired upon detection of associated abnormal situations. This significantly reduces the unnecessary management traffic and the load on central stations, thus reducing the scalability barrier.

5.2.2 Performance

We illustrate the performance characteristics of JEMS by comparing the performance of a monitoring application using SNMP to that of an application that has the same functionality but was implemented on our prototype system.

Consider an SNMP-based monitoring application that involves *n* MIB variables. The cost function of the overall response time of this application can be approximated by

$$T_{SNMP} = n \cdot T_D(SNMP\_Get) + T_C(manager) + T_{MIB}$$

$T_D(SNMP\_Get)$ represents the delay of each SNMP Get request between the centralized station and the SNMP agent, $T_C(manager)$ represents the computation time of the application at the management station, and T_{MIB} represents the total time spent by the SNMP agent in MIB searching during one invocation of the application. $T_D(SNMP\_Get)$ depends on the round-trip transmission delay of the SNMP Get/Get-Response message pair, plus message processing time at both ends.

Consider a JEMS implementation of the same application. The functionality of the monitoring application can be encapsulated into an Operator object, which is in turn bound with a IMO delegated to the managed device, and a get operation on the IMO's result attribute would return the computation result. After the IMO is delegated and ready to serve, the typical response time cost function for one get operation can be approximated by

$$T_{JEMS} = T_D(RDI\_Get) + n \cdot T_{LD}(SNMP\_Get) + T_C(agent) + T_{MIB}$$

$T_D(RDI\_Get)$ represents the delay of each RDI get operation between the management station and the JEMS server, $T_{LD}(SNMP\_Get)$ represents the local delay of each SNMP Get message between the SNMP accessor and the SNMP agent, $T_C(agent)$ represents the computation time of the Operator object in the managed device, and T_{MIB} represents the total time spent by the SNMP agent in MIB searching.

In a Wide Area Network (WAN), message transmission delays constitute the major part of both $T_D(SNMP\_Get)$ and $T_D(RDI\_Get)$, therefore we can assume

$$T_D(SNMP\_Get) \approx T_D(RDI\_Get)$$

$T_C(manager)$ and $T_C(agent)$ are influenced by system parameters, such as CPU speed and memory capacity, of the management station and JEMS server respectively. Although the centralized station is almost always much more powerful than the managed device (or the proxy machine) where the JEMS server is running, it is almost always shared but multiple tasks as well. When we consider simple monitoring applications, we can safely assume that $T_C(manager)$ and $T_C(agent)$ are of the same order of magnitude. With delegated objects, we have moved the network polling to local MIB access, thus the delay associated with SNMP Get requests is very small, i.e.,

$$T_{LD}(SNMP\_Get) << T_D(SNMP\_Get)$$

Therefore, the residual performance difference is

$$T_{SNMP} - T_{JEMS} = (n-1) \cdot T_D(SNMP\_Get) + d$$

Of course, the initial delegation of a IMO to the JEMS server consists of such complex steps as RDI create operation request, classes download, objects instantiation and initialization, and may take quite some time. However, once the delegation is completed and the "cold start" cost is paid, intelligent monitoring continuously beats SNMP-based method, so it only takes a few more program invocations to amortize the initial cost. And in a WAN with large delay and for applications involving many MIB variables, this amortization will be even faster.

5.2.3 Extensibility

The most outstanding benefit of JEMS system is that it provides online extensibility that traditional systems don't have. Once up and running, a device agent is equipped with a fixed set of functionality. With a JEMS running in the device, its functions can be dynamically extended/changed without having to bring the system offline.

For example, there are some predefined trap variables that SNMP agents use to report simple abnormal situations in the device. However, the number of these variables is fixed and their semantics are static. If later a new situation is identified and demands attention, the only way to incorporate it into the agent is to add a corresponding trap variable to the MIB and recompile the agent. During this process, the agent has to be stopped and its service interrupted.

With a JEMS running in the device, things become easier and more flexible. First, with an appropriate Operator bound to it, an Analyzer needs to be delegated to the device to monitor the potential alarm. Then, when we believe that the symptoms associated with the alarm has changed, the Operator can be properly reconfigured to reflect the change. For instance, if an OpThreshold operator was adopted, we may want to reset the values of its upper and lower thresholds. To go even further, if a new alarm situation arises that can not be taken care of by any Operator currently available, we can always write a new Operator class that identifies the situation, and then bind an instance to the Analyzer. The whole process can be accomplished without suspending the agents' current functionality.

5.3 Conclusions

JEMS provides a simple and flexible model to construct monitoring systems, by allowing dynamic creation, manipulation and integration of delegated monitoring objects. Network managers can use the predefined IMOs provided by network device vendors, and their own objects to build distributed and dynamic management applications. By taking advantage of IMOs running in the managed devices, the intelligent monitoring system has better scalability, performance and online extensibility than centralized polling systems.

Furthermore, since JEMS is based on standard industry-proven Java technologies, it is easier to implement and has better portability than MbD [19]. Unlike the mobile agents method [13], the change to monitoring paradigm introduced by JEMS is incremental rather than fundamental, making its integration with current polling-centric systems much easier.

Chapter 6 Conclusions

With increase in the complexity of modern communication networks, it is imperative that there be commensurate advances in the tools and techniques used to manage these networks. However, as discussed earlier, conventional network monitoring and management systems rely on a framework and related techniques that have inherent drawbacks. This thesis has presented the work that we have done to facilitate intelligent network monitoring based on the Java technology. We are trying to draw some conclusions in this last chapter as follows.

Chapter Organization

Section 6.1 summarizes the work presented in the previous chapters.

Section 6.2 looks at possible future improvements that could be made to the current design and implementation.

6.1 Summary

Concurrent network monitoring systems adopt a centralized framework where most of the monitoring intelligence and processing burdens rest at the manager applications running in a central station. This poses several major problems.

Since all the monitoring interactions and processing have to go through the management station, it becomes the bottleneck and single point of failure, leading to a system that is difficult to scale up.

Manager applications can only interact with network elements through low-level general-purpose interfaces such as SNMP, huge volume of raw data have to be transferred to the management station, which causes high communication overhead and significant delay, known as the micro-management problem.

The set of services offered by the element agents is fixed and statically instrumented, which hinders the cost-effective extension and improvement of monitoring systems.

To tackle these problems, we have basically done three things: (1) brought forward the "intelligent network monitoring" concept; (2) according to this concept, proposed a Java-based architecture, JEMS; (3) implemented a prototype JEMS system, and validated its efficiency.

6.1.1 Intelligent Network Monitoring

The concept of Intelligent Network Monitoring is comprised of two elements: distribution of intelligence and dynamic agent extensibility:

Distribution of Intelligence. Instead of bringing data from the devices to the central station, parts of the monitoring applications themselves, encapsulated in various objects, are *distributed* or delegated to and running in the managed devices. The manager host and the network as a whole can then be relieved from the bottleneck and the micro-management problems.

Dynamic Agent Extensibility. Through a public calling interface over the network, the manager application can remotely distribute/remove such objects to/from a network device whenever it likes; the code required to manipulate these objects may be obtained and linked to the device agent on demand, making it truly *dynamically extensible*.

6.1.2 Intelligent Monitoring via JEMS

Java-based Extensible Management Server (JEMS) is the architecture we proposed to facilitate intelligent network monitoring. It consists of two parts:

The JEMS Server. A Java-based element agen that supports distributed and dynamic network management. It runs as a server process in the managed network element and consists of the Remote Delegation Interface (RDI) and a runtime environment that implements the RDI.

Intelligent Monitoring Objects (IMOs). Specialized Java objects that manager applications delegate to the JEMS via RDI calls. With specific functions encapsulated, these objects perform network monitoring right in the managed devices. IMOs together with the underlying JEMS provide for a flexible, easy-to-program, and highly reusable intelligent monitoring system.

6.1.3 Intelligent Monitoring Validated

A prototype system was implemented based on the JEMS architecture. And simple yet typical experiments have validated the advantages of intelligent network monitoring over traditional polling-based schemes.

By distributing monitoring intelligence closer to where the information to be processed is located, our system significatly reduces the network traffic and delay incurred between managers and agents, and eliminates the bottleneck and single point of failure problems existing in traditional network management systems. On the other hand, the dynamic extensibility of our system allows managers to extend agents' ability accordingly as network management requirements evolve. Also, manager applications can recognize and take advantage of the difference in resource availability of various network devices, and make proper tradeoffs between computation and communication cost.

6.2 Future Work

There are some improvements and enhancements that could possibly be made to the JEMS architecture and added to its prototype implementation.

6.2.1 User Authentication, Access Control and Privacy

In our current design, accessing to the RDI interface and services is not controlled, anyone having the DSP's stub object can manipulate objects in the JEMS in whatever way they want. A possible remedy to this is adding an authentication scheme to the RDI interface. This may include a new *authentication* operation plus modifications to all the existing interface operations. The authentication operation takes a manager’s name and password and returns an authenticator object if the manager’s identity is correctly verified. To access normal delegation operations, the remote manager has to first authenticate itself to the DSP, obtains an authenticator object and later includes it in every remote method call into the DSP. The authentication mechanism has to be designed in such a way that it prevents replay attacks.

Once a manager is authenticated to the DSP, the latter decides what access privileges the user is allowed, and only executes those requests that are conformant to its access privileges. To this end, a proper access control model, together with necessary data structures and control mechanism, has to be adopted and enforced.

If the monitoring data transferred via the network includes business-sensitive information, we may use a customized socket layer for the RMI calls, by subclassing the java.rmi.server.RMISocketFactory class to implement a secure transport.

6.2.2 Manager-side APIs and Tools

Although the JEMS architecture provides quite a complete agent interface, it has not addressed how those interface methods can be collectively used to write monitoring applications, namely we lack manager-side APIs and tools for application authoring. Therefore, in addition to making improvements and enhancements to the agent interface, we may want to:

(1) Use Java’s component technology to write JavaBeans that implement varous high-level manager-side management tasks (which involve multiple IMOs and a series of interactions with them). The public methods exported by these

JavaBeans form the API for manager applications development.

(2) Write a GUI-based developer tool for visual application programming using those JavaBeans.

(3) Prepare a deployment tool that facilitates the packaging and installation of manager applications and their resources over the network.

6.2.3 Intelligent Network Control

Using JEMS based architecture to do network control should be a natural continuation of the research work. We would like to identify the most commonly needed network control functions, encapsulate them in corresponding *Intelligent Control Objects* (ICOs) classes using the Operand-Operator Paradigm, and deploy these classes and their associated Operator classes to the bytecode server, so that they become available for download to the JEMS-ready network devices upon relevant RDI invocations from the manager. Network control operations will thus be performed from within the devices themselves, providing for "on-spot" handling of various alarms and events. Furthermore, for the same reasons with IMOs, the behavior of ICOs can be dynamically configured and enhanced, which, together with IMOs, facilitates a flexible intelligent network management system.

6.2.4 Autonomous Management Domain with Jini

In a situation where the managed devices come online and go offline irregularly and asynchronously, e.g., VSATs in a satellite network, automatic detection and monitoring of these devices might be desired. Jini connection technology [17], built on top of Java, stands out as an ideal choice for us to incorporate that into the JEMS architecture.

Jini provides simple mechanisms which enable devices to plug together to form an *impromptu community* – a community put together without any planning, installation, or human intervention. Each device provides services that other devices in the community may use. These devices provide their own interfaces, which ensures reliability and compatibility. However, we had better wait until after Sun Microsystems officially merges Jini into the Java 2 Micro Edition (J2ME) [25], when we can evaluate how this will affect the size of the core API libraries and the Java Virtual Machine (a.k.a. *KVM* – *K*ilobyte J*VM*) that comes with J2ME.

6.2.5 Hierarchical Intelligent Network Management System

Because of its intrinsic layered design, the JEMS architecture can be extended to a hierarchical structure to accomodate the logical domains of the underlying managed network. In each sublayer between the network control center and the network agents, subnetwork or domain managers can be equipped with JEMS based servers that act in two roles: as a manager of the stations in the immediate lower layer, and as an agent for its upper layer station. In such a middle-layer management station, the MIB Accessor will be implemented to use RDI invocations (instead of SNMP protocol) to communicate with lower-layer stations, and INFOs in the MIT will use the data acquired through the MIB Accessor to maintain an object-oriented abstraction of the subnetwork/domain the management station is in charge of.

Appendices

portNumber	Identification of the port	Integer
portMaxBandwidth	The maximum incoming bandwidth of the port (cells/s)	Integer
portMaxBandwidthOut	The maximum outgoing bandwidth of the port (cells/s)	Integer
portCDVT	The Cell Delay Variation Tolerance associated with this physical port. Connections take their default value for CDVT from the input side port	Integer
portVbrOverbooking	The percentage of overbooking for VBR connections. The default value is 100 (no overbooking)	Integer (1..500)
portVbrBufferOverb	The percentage of buffer overbooking for VBR connections. The default value is 100 (no overbooking).	Integer (1..500)
hwPortBufferSize	The logical size of this port's output buffer, in cells	Integer
portAllocBandwidthIn	The allocated incoming bandwidth of this port (cells/s)	Gauge
portUsedBandwidthIn	The incoming bandwidth being used on this port (cells/s)	Gauge
portReceivedCells	The number of cells received on this port	Gauge
portAllocBandwidthOut	The allocated outgoing bandwidth of this port (cells/s)	Gauge
portUsedBandwidthOut	The outgoing bandwidth being used on this port (cells/s)	Gauge
portTransmittedCells	The number of cells transmitted on this port	Counter
hwPortQueueLength	The number of cells in this port's output buffer	Gauge
hwPortOverflows	The number of seconds in which cells were dropped because this port's output buffer was full	Counter

Table 11: Port-related SNMP Variables

q2931StatsPort	The value of this variable identifies the port of this signaling path	Integer
q2931StatsVPI	The value of this variable identifies the VPI of this signaling path statistics entry	Integer
q2931CallsCompletions	The number of successfully completed calls on this signaling path	Counter
q2931CallsFailures	The number of call failures on this signaling path	Counter
q2931CallsRejections	The number of connections on this signaling path that were rejected by the far end	Counter

Table 12: Signaling-related SNMP Variables

pathPort	Indentification of the input port which contains this path	Integer
pathVPI	The VPI (Virtual Path Identifier) of this path	Integer
pathMaxBandwidth	The maximum bandwidth of this path (cells/s)	Integer
pathAllocBandwidth	The allocated bandwidth of this path (cells/s)	Gauge
pathUsedBandwidth	The bandwidth being used on this path (cells/s)	Gauge
pathCells	The number of cells transferred over this path	Counter
pathUptime	The elapsed time since this path was created	Time Ticks
pathRejectedCells	The number of cells over this path that were rejected or dropped by the policer on the switch fabric	Counter

Table 13: Incoming Path-related SNMP Variables

opathPort	Indentification of the input port which contains this path	Integer
opathVPI	The VPI (Virtual Path Identifier) of this path	Integer
opathMaxBandwidth	The maximum bandwidth of this path (cells/s)	Integer
opathAllocBandwidth	The allocated bandwidth of this path (cells/s)	Gauge
opathUsedBandwidth	The bandwidth being used on this path (cells/s)	Gauge
opathCells	The number of cells transferred over this path	Counter
opathUptime	The elapsed time since this path was created	Time Ticks
opathRejectedCells	The number of cells over this path that were rejected or dropped by the policer on the switch fabric	Counter
opathVbrOverbooking	The percentage of overbooking for VBR connections. The default value is 100 (no overbooking)	Integer (1..500)
opathVbrBufferOverb	The percentage of buffer overbooking for VBR connections. The default value is 100 (no overbooking)	Integer (1..500)

Table 14: Outgoing Path-related SNMP Variables

References

[1] W. Stallings, SNMP, SNMPv2 and CMIP: the practical guide to network management standards, Addison-Wesley, Reading, Mass., 1993.

[2] W. Stallings, SNMP, SNMPv2 and RMON: practical network management, Addison-Wesley, Reading, Mass., 1996.

[3] J. Rumbaugh et al., Object-oriented modeling & design, Prentice Hall, 1991.

[4] Gopalan S. Raj, "A Detailed Comparison of CORBA, DCOM and Java/RMI," http://www.execpc.com/~gopalan/misc/compare.html.

[5] N. J. Muller, "Translation of GDMO/ASN.1 to Java objects for network management," Proceedings of the 1998 IEEE International Conference on Communications (ICC'98), Atlanta, GA, 1998, vol.2, pp. 1140-1144.

[6] GDMO and ASN.1 Compiler and Code Generator for Java, C, C++, http://www.technovision.dk/~uh/prod03.html.

[7] Sun Microsystems, "The Java Language Environment," http://java.sun.com/docs/white/langenv/, 1996-1999.

[8] Sun Microsystems, "The Java Platform," http://java.sun.com/docs/white/platform/javaplatform.doc.html, 1996-1999.

[9] Sun Microsystems, "Java 1.1.x Application Programming Interface," http://java.sun.com/products/jdk/1.1/docs/api/packages.html, 1996-1999.

[10] Sun Microsystems, "Java Remote Method Invocation," http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/, 1996-1999.

[11] AdventNet Inc., "Adventnet SNMP Java API," http://www.adventnet.com/, 1996-1999.

[12] G. Pavlou et al., "Distributed intelligent monitoring and reporting facilities," *Distributed Systems Engineering*, vol.3, no.2, pp. 124-135.

[13] M. Baldi et al., "Exploiting code mobility in decentralized and flexible network management," Proceedings of the 1st International Workshop on Mobile Agents (MA'97), pp. 13-26.

[14] N. Anerousis, "An information model for generating computed views of management information," Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Newark, DE, October 1998.

[15] N. Anerousis, "An architecture for building scalable, Web-based management services," *Journal of Network and Systems Management*, vol.7, no.1, pp. 73-104.

[16] M. Leppinen et al., "Java- and CORBA-based network management," *Computer*, June 1997, vol.30, no.6, pp. 83-87.

[17] Sun Microsystems, "Jini Technology," http://www.sun.com/jini/, 1998-1999.

[18] IETF, "Management Information Base for Network Management of TCP/IP-based Internets: MIB-II," ftp://ftp.isi.edu/in-notes/rfc1213.txt/, March 1991.

[19] G. Goldszmidt and Y. Yemini, "Distributed management by delegation," Proceedings of the 15th International Conference on Distributed Computing, June 1995.

[20] E. H. Mamdani, R. Smith and J. Callaghan, *The Management of Telecommunication Networks*, Ellis Horwood Limited, 1993.

[21] S. Yemini, E. Moses, Y. Yemini and D. Ohsie, High speed and robust event correlation, *IEEE Communications Magazine*, May 1996.

[22] J. E. White, Mobile agents, *Software Agents*, MIT Press, 1996.

[23] R. S. Gray, Agent Tcl: a transportable agent system, Proceedings of the CIKM'95 Workshop on Intelligent Information Agents.

[24] Sun Microsystems, "Java Naming and Directory Interface," http://java.sun.com/jndi/, 1999

[25] Sun Microsystems, "Java 2 Platform, Micro Edition," http://java.sun.com/j2me/, 1999.