Distributed Management by Delegation

Germán S. Goldszmidt

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
1996

ABSTRACT

Distributed Management by Delegation

Germán S. Goldszmidt

Network delays are becoming the most critical performance problem for distributed applications. Traditional client server interactions do not scale well to environments where those delays are relatively long. *Elastic processes* are executing programs that can dynamically integrate new functionality sent from external processes as *delegated agents*. Elastic applications overcome network delays by dynamically moving computations closer to the resources that they need to access. Delegated agents are written in arbitrary programming languages, and their execution can be remotely controlled. Elasticity defines an application-level interprocess exchange of code, dynamic loading with multithreaded execution, and remote control. The elastic processing architecture extends dynamic linking of delegated agents across remote computers.

Current network management systems follow a platform-centric, static software paradigm that allocates most responsibilities to platform-based hosts, and leaves network devices with minor service support roles. This paradigm results in inefficient allocation of management responsibilities and intrinsically unreliable systems. It forces management applications to micro-manage devices, and results in failure-prone management bottlenecks, and limitations for real time responsiveness. The dissertation presents a more flexible management paradigm, namely *Management by Delegation* (MbD). MbD permits programmable extensibility of network functions to address rapidly changing network environments. It improves the reliability and availability of networked systems by dynamically embedding in them the intelligence required for autonomous self-management.

Contents

List of Figures

To Orna and Jonathan

Acknowledgements

I have been very fortunate in having Yechiam Yemini as my dissertation advisor. He inspired my interest in network and system management, taught me the essence and principles of good research, and guided me through the completion of this dissertation. Working with "YY" was a challenging and rewarding experience. I would also like to thank Sal Stolfo, Leana Golubchik, Mike Bauer, and Joe Hellerstein, for having served on my dissertation committee.

Many of the software components described in the dissertation were implemented as research projects by Sharon Barkai, Antonis Maragkos, Jim Tuller, and Chris Wood. Alex Dupuy gave us a lot of great system advice. Cristina Aurrecoechea and Patricia Soares reviewed the thesis draft and provided valuable comments. Susan Keil and Kyra Voss made sure that the English was accurate. I had many interesting exchanges and collaborations with many colleagues, both in person and via e-mail. A partial list of these would include Amatzia Ben Artsi, Karl Auerbach, Joe Betser, Mike Erlinger, Mark Kennedy, Kraig Meyer, and Jacob Slonim.

I am greatful to all the members of the Distributed Computing and Communications (DCC) Laboratory for their interest in my work, their useful suggestions, and for making the lab such a great place. In particular, Jakka Sairamesh was a wonderful officemate, Patricia Soares and Danilo Florissi were always very helpful and friendly, and Ron Erlich kept our systems running.

Faculty and staff members of the Computer Science Department have generously provided advice and assistance during my tenure at Columbia. In particular, thanks to Al Aho, Zvi Galil, and Sal Stolfo for their good advice, and also to Dan Duchamp for participating in my thesis proposal committee. The staff members of the department, Germaine L'Eveque, Mel Francis, Martha Peres, Susan Tritto, Renata Valencia, and Mary Van Starrex, were always dedicated, helpful and friendly.

I am also very thankful to my managers and colleagues at the IBM T.J. Watson Research Center. Shaula Yemini invited me to IBM Research and encouraged me to start my PhD studies. Josh Auerbach, Jim Russell, and Shaula Yemini gave me the opportunity to pursue my dissertation while working part time at IBM. Stu Feldman, Pat Goldberg, Brent Hailpern, Jim McGroddy, Dave McQueeney, Scott Penberthy, Danny Sabbah, and Zvi Weiss have helped me over the years in various ways at IBM.

Finally, I would like to thank my wife Orna and our son Jonathan for their support, endurance and patience. They always remind me of what is truly important.

1

Introduction

This dissertation introduces a novel technology for mobile agent computing, *elastic processing*. A *mobile* agent is a program that is dynamically dispatched to, and executed at, a remote site. For example, mobile agents can be dispatched to (1) remote commerce servers to search for and book a vacation; (2) a remote TV set-top box to program it to decode compressed video; (3) a remote network element, such as a router or a switch, to control its operations. Many emerging networked applications need such technologies to dynamically deploy software at distributed computing devices.

Recent proposals for mobile agent technologies are based on specialized interpreted languages, like Java [Gosling and McGilton, 1995] or Telescript [White, 1994]. In these frameworks, an agent is a script that is dispatched to and executed at a remote interpreter. In contrast, this dissertation introduces a language-independent agent technology. Delegated agents can be coded in an arbitrary programming language, compiled or interpreted. They are dispatched to remote elastic processes and are dynamically linked with them and executed as threads under remote control.

Elastic processing offers significant advantages over the above approaches. First, it supports a much broader class of applications than language-based scripting agents. For example, scripted agents are not appropriate for handling real-time computations such as video decoding. Second, it builds on existing general purpose programming languages and their development support. Third, one can delegate an entire interpreter to an elastic process and then delegate script agents to it. Therefore, elastic processing can handle language-based agents as a specialized case.

This thesis also introduces a novel approach to managing networked systems, using elastic processing. Effective manageability of distributed systems is a critical need for all modern organizations. A *Network Management System* handles problems related to the reliability, configurability, accountability, efficiency, and security of heterogeneous distributed computing environments. The practical handling and resolution of many challenging technical problems fall under the aegis of network and system management. Network management is concerned with monitoring, analysis and control of network behaviors to assure smooth network operations.

Operational data is collected by instrumentation of network elements and organized in *Management Information Bases* (MIBs). For example, a router MIB can include counters for various traffic statistics at the router ports. This operational data is gathered by a centralized 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 labor-intensive management paradigm does not scale for the size and complexity of emerging networked systems. Hence, new technologies are needed to automate and decentralize management functions.

Management-by-Delegation (M_bD) is a novel network management paradigm that applies elastic processing technologies to address many of the problems of current network management systems. Management applications delegate mobile agents to embedded elastic processes at network elements and domains. These delegated programs automate the monitoring, analysis and control of network elements. For example, delegated programs can monitor a large number of MIB variables to detect when a domain experiences problems. These agents can invoke delegated diagnostic programs and then execute delegated policies to handle the problems. Management responsibilities can thus be decentralized and automated.

1.1 Contributions

The thesis defended by this dissertation is that elastic processing addresses many of the needs of emerging distributed environments, and, in particular, those of network management systems. Some of the results presented in this dissertation have been reported in the following articles [Goldszmidt *et al.*, 1991; Yemini *et al.*, 1991; Goldszmidt and Yemini, 1991; Goldszmidt, 1992; Goldszmidt and Yemini, 1993; Goldszmidt, 1993a; Goldszmidt, 1993b; Goldszmidt and Yemini, 1995]. The contributions presented in this dissertation include:

- Elastic Processing, a novel model of distributed computing interaction. Elastic processing includes (1) a Remote Delegation Service (RDS) to dispatch agents to a remote elastic process, invoke them as threads and control their execution, and (2) a runtime architecture supporting dynamic linking, multithreaded execution, authentication based security, and remote control of delegated agents.

- Management by Delegation, a novel framework for distributed network and systems management. M_bD dynamically distributes network management computations to elastic servers at the network devices where the managed resources are located. M_bD is integrated within current management protocols like SNMP.

- A formal model used to describe the behaviors of managed entities and their observations by management applications. This model elucidates many of the advantages of M_bD over current network management paradigms. We design

and implement a distributed management application to compress operational data by computing *health* index functions.

- A new framework to support the dynamic definition of external data models at the networked devices. It consists of a specification language for views and runtime extensions to the M_bD-server. User defined views are applied to filter device data, execute relational operations, provide access control and atomic semantics.

Chapter Organization

Section 1.2 outlines elastic processing with delegated agents.

Section 1.3 outlines network management issues.

Section 1.4 outlines *Management by Delegation*,

Section 1.5 briefly outlines two management applications of M_bD.

Section 1.6 presents a roadmap to the remaining chapters of this thesis.

1.2 Elastic Processing with Delegated Agents

An elastic process is a multithreaded process linked with the implementation of an elastic runtime library. This is analogous to an RPC process which must be linked to an appropriate RPC runtime library. The elastic-processing runtime library supports multithreaded dynamic linking and loading of delegated agents, remote communications with agent instances, and remote control of the agents' execution.

The *program code* and the *process state* of an elastic process can be modified, extended, and/or contracted during its execution. New agent code can be added and instantiated as a thread inside an executing elastic process address space. These changes, which are internal to the elastic process, result from an explicit interaction with another process, namely *remote delegation*. The main action of remote delegation is the transfer of an agent's code to an elastic process. Subsequent actions allow the instantiation of an agent and control of its execution inside the elastic process.

The technologies developed to support elastic processing consist of a remote delegation protocol service and a runtime architecture for elastic processes. Processes use the *Remote Delegation Service* (RDS) to configure and control elastic processes, and to communicate with agents. A delegator process can instantiate, suspend, resume, abort, and remove a delegated agent. We call the runtime environment of an elastic process the *"Delegation Backplane Middleware"* (DBM). The DBM allows the elastic process to be dynamically extended and contracted. DBM supports translation and dynamic linking of delegated code, a multithreaded execution environment, and internal and external communications.

1.2.1 Elastic Processing has many Applications

Consider a laptop computer application to make reservations for a vacation, including hotel, car, tours, and so forth. This application will support arbitrary personal constraints which are defined by rules to ensure the best possible vacation for a given budget. Assume that the relevant information is available at some remote *Web* server. A non-elastic Web server allows its clients to retrieve information based on predefined queries, e.g., using forms. However, the Web server could not have predefined all the possible types of queries for each vacation. The application will need to retrieve large amounts of data from the server, filter it locally at the laptop host, and then execute the booking transaction at the server.

This scheme is inefficient, insecure and expensive. It is inefficient since it wastes CPU cycles of both client and server hosts. By the time the information has been retrieved and filtered, it may no longer be relevant, e.g., the hotel room may have been taken. It is insecure because the servers expose the client to a lot of data which the data owners may prefer to keep private for competitive business reasons. If this transaction is being performed over a wireless or long-distance phone line, the cost of the data exchange is expensive.

An alternative solution is to dispatch delegated agents to an elastic Web server. The delegated agent will contain all the specific rules and constraints for a particular transaction. Such an agent could be delegated to servers from different organizations, prompting them to compete on real time. Using delegated agents, applications can reduce their consumption of resources, e.g., network delays and expenses are avoided by reducing the transfer of unnecessary data. Thus, elastic processing can be used to improve the performance of applications that execute in computer hosts with insufficient computing resources or low bandwidth networks. Section 2.6 describes the use of elastic processing to extend executing applications dynamically, support interoperability via application gateways, adapt to varying computational resource availability, and upgrade, customize, and monitor executing software.

1.2.2 Elasticity Presents Efficient Performance Tradeoffs

Most distributed applications and systems are organized following a Client-Server (C/S) interaction paradigm. For instance, many distributed applications are implemented using RPC. The traditional C/S paradigm does not perform well in environments where network delays are relatively large. As computer processors get increasingly faster, distributed applications processes will spend an increasingly larger portion of their cycles *idle-waiting* for remote interactions. Network latency will then become the most significant bottleneck for distributed applications. Network latency is primarily bound by the network's topology, its routing mechanisms, and application quality of service requirements. Even if all of these could be optimized, there would still be the fundamental barrier of the speed of light on the transfer medium. Elasticity is an *"inverse-caching"* solution to the network latency bottleneck, i.e., is moving application code closer to the location of its data. Distributed applications

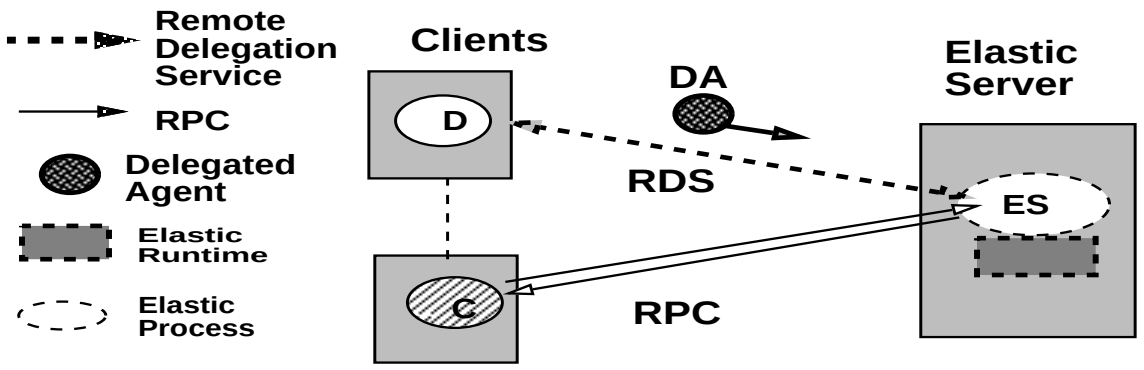

Figure 1.1: Delegating Mobile Agents to an Elastic Server

use delegated agents to reduce the number of remote C/S interactions.

Elastic processing supports and extends alternative distributed programming models such as RPC. For example, an application can use RDS to extend the services of an elastic server process. Consider the following C code fragment that contains a sequence of RPC calls to a remote file server:

```
while (i=0;i<n;i++){
 rpc-get-info (&r);
 rpc-combine(p, r)
}
```

A process can delegate this code fragment and install it as a new procedure in the elastic server. Thus a code fragment that required $2n$ RPC invocations can be performed with only one remote interaction. Figure 1.1 shows a process, D, delegating an agent, DA, to an elastic server process, ES. After ES dynamically links the code of DA it can be invoked by D and other remote clients. Thus an elastic server process can adapt to provide its clients with dynamic extensions of its functionality.

Implementations of Elastic Processing

We implemented the first elastic process prototype as an extensible server for management applications [Goldszmidt *et al.*, 1991]. This initial application included a multithreaded server and several clients. *System Management Arts* implemented an enhanced product version, the SMARTS Operations Server (SOS) [Dupuy, 1995].

1.2.3 Related Work

Distributed applications are designed and implemented following diverse remote communication models. Examples of implementations of these models include several "remote-" prefixed mechanisms, such as Remote Procedure Call (RPC), remote execution, remote evaluation, and remote scripting. Section 2.7 compares these

models with remote delegation to elastic processes. The following paragraphs outline some of the aforementioned comparisons.

Comparison with RPC

One of the most popular distributed application programming mechanisms is RPC. RPC server designers must predict and code the entire range of services for which a particular server may be invoked. For many distributed applications, however, it is typically not possible to predict at application design time all the possible scenarios in which a server can be involved. The definition and binding of a delegated agent procedure can be done at the latest possible moment, just before its use.

Sometimes the invocation of a server's procedure should be tied to events that are not under the client's control. For instance, the occurrence of an alarm in a device should trigger the execution of a server procedure to handle it. Using an RPC mechanism would block the client until the occurrence of such an event and then until the completion of the RPC. In contrast, the execution of a delegated agent in an elastic server can be triggered by independent event occurrences in the server's host, asynchronously with the client's execution.

Remote Scripting with Agents

Several programming languages have recently been proposed to write *scripting* or *mobile* agents. Examples of these languages are Java [Gosling and McGilton, 1995], Safe-TCL [Borenstein, 1994], and Telescript [White, 1994]. Section 2.7.4 describes these technologies, and Section 2.7.5 presents a more detailed comparison with the work presented here. A detailed taxonomy is presented in Table 2.1. The following paragraphs briefly outline some of their differences.

The above scripting technologies are language-based mechanisms. RDS is a generic, language neutral mechanism for dynamically extending processes under remote control. Delegated agents have been written in several languages. An elastic process can be dynamically extended with a new interpreter for a scripting programming language. The process will then be able to accept delegated agents written in that language. Delegated agents can be compiled or interpreted, while remote scripting agents are always interpreted. Many tasks cannot be effectively handled by safe interpreted languages. For instance, Telescript agents cannot directly examine or modify the physical resources of the computers on which they execute. Similar restrictions apply to other *"safe"* languages, like Safe-TCL and Java. These limitations all but eliminate these languages as potential technologies for the many applications which require such facilities. Delegated agents can execute as threads with explicit access to the underlying physical resources.

Elastic processes permit explicit remote control of the execution of delegated agents to authorized parties. Script interpreters do not provide explicit support for remote control of their scripts. Remote evaluation and scripting combine code transfer and its execution in one action. Elastic processing separates them into independent

actions. Thus RDS provides a fine granularity of access control to delegated agents and the threads instantiated from them.

Elastic processes can be configured and customized to support arbitrary security and safety policies. Remote scripting technologies typically enforce a pre-defined *"one-size-fits-all"* security policy. Some remote scripting languages require a reliable transport connection to exchange agents, e.g., Telescript. RDS can execute over both reliable (TCP) and unreliable (UDP) transport protocols. This is very important for many distributed applications, and in particular for network management. For instance, a network fault diagnosis application should not rely on functioning transport connections.

1.3 Network Operations and Management

Network administrators and operators that manage large distributed systems need automated tools to maintain seemly operating networks. Before 1990, most available management tools were used in an ad-hoc fashion. At the time, the few generic tools available provided very basic functions. For instance, operators used `traceroute` [Jacobson, 1988] to track the route that IP packets follow or to find a "miscreant" gateway that is discarding some packets. Some networks supported more sophisticated, but proprietary, management tools.

As networks grew in size and became more complex, their operational costs increased substantially. Network operators had to become adept at handling non-amenable problems in real-time for an ever growing melange of devices. Yet most useful tools were vendor-specific and supported only a certain class of devices. The increased complexity of operations created a demand for common, vendor-neutral, interoperable, and integrated solutions. Standards organizations ameliorated this situation by providing management interoperability frameworks such as the IETF's SNMP [Case *et al.*, 1990] and the ISO's CMIP [ISO, 1990a].

Network Management Goals

Network management aims include the detection and handling of faults (e.g., network cleavages), performance inefficiencies (e.g., high latency delays), and security compromises (e.g., unauthorized access). To accomplish these goals, management applications do the following:

- Collect real time data from network elements, such as routers, switches, and workstations. For example, they collect the number of packets handled by the given interface of a router.
- Interpret and analyze the data collected. For instance, they may recognize security events, such as repeated illegal attempts to login on a workstation.
- Present this information to authorized network operators, possibly by displaying a map of current traffic.

- Proactively react, in real time, to management problems, possibly by disabling a link that is experiencing faults.

These activities are organized in accordance with guidelines established by network management standards. See Appendix A for a more detailed overview and critical analysis of network management standards.

1.3.1 Components of a Network Management System

Network management components are classified into *platform managers* and *device servers* or "agents"[1]. Figure 1.2 shows a diagram of the organization of a typical network management system. It consists of a management station with a console for the operators, and management "agents" (device servers) embedded in network elements. The management station and operators are located at the *Network Operating Center*, NOC. Applications in the management station assume a *manager* role. Typically, an "umbrella" application provides a *Graphical User Interface* (GUI) that integrates the management applications for the network operators.

These applications execute concomitantly with the GUI to perform specific management functions, e.g., accounting or security. Device servers are software processes embedded within each manageable network entity. These servers collect device data in *Management Information Bases* (MIBs) and support a management protocol. For instance, an SNMP-agent is a device server that implements an SNMP MIB and responds to SNMP requests.

Manager applications use a management protocol to exchange data to and from an MIB. For example, an SNMP-agent within a router collects information about network traffic and routing tables and organizes it in an SNMP MIB. A management application in the platform can retrieve this data using the SNMP `Get` command. The retrieved data is then filtered and displayed graphically on a network traffic map. A manager application can also use SNMP to perform control operations over a networked device. For example, the application can use SNMP's `Set` to modify the value of an MIB variable to 0. The device server at the device can be programmed to recognize this action as a request to reboot the device.

SNMP Allocates Most Processing to the NOC Hosts

The proper allocation of responsibilities between the centralized management platforms and the network elements is critical for accomplishing effective manageability. Current network management systems follow a platform-centric framework that allocates most responsibilities to the management applications executing at the NOC platform. Mostly menial tasks are performed by the device servers, e.g., SNMP-agents.

[1]Unfortunately, the network management community uses the term "agent" to refer to a device server. This management "agent" is a stationary server process that supports a network management protocol and performs certain tasks. We will either use the term *device server* or qualify the name, e.g., SNMP-agent, to avoid confusing them with delegated agents.

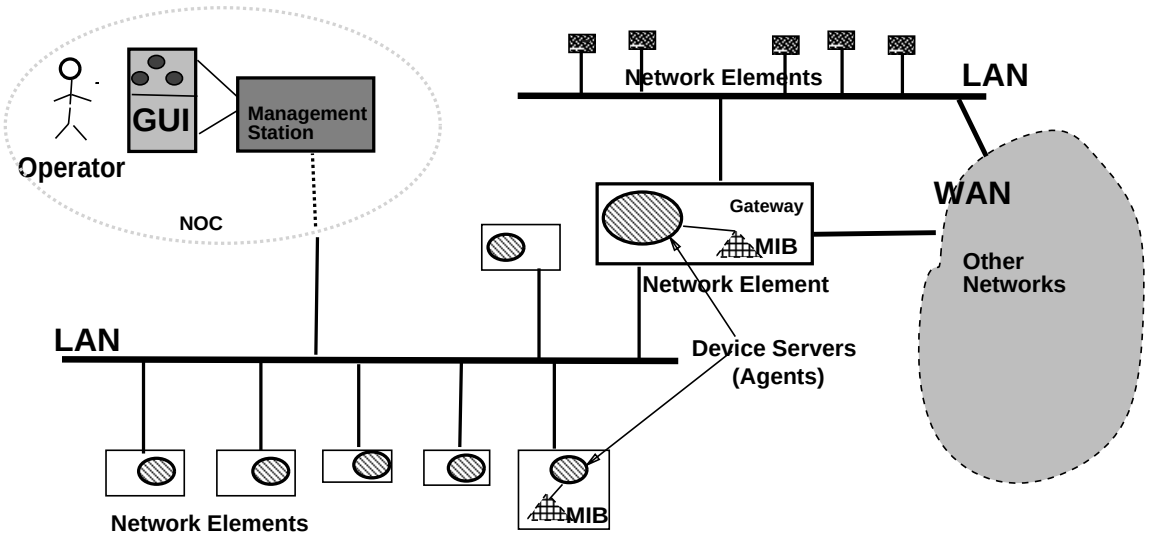

Figure 1.2: Components of a Network Management System

The explicit assumption is that network elements cannot afford to be "intelligent", but must rely on the centralized smarts of the platform-based applications. The SNMP framework, for instance, assumes that network devices have limited computing resources available, and hence SNMP-agents should only perform a minimal set of duties.

Management applications are implemented following a traditional C/S model The service interfaces and the structure of MIBs are strictly defined by standards. The implementations of these processes are statically compiled and linked. A client process in a manager role can only invoke a *fixed* set of predefined services. This set cannot be modified or expanded without the recompilation, reinstallation, and reinstantiation of the server process. This rigid division of functionality hinders the development of effective management systems.

1.3.2 Centralization Induces Performance Limitations

Platform-centric management systems establish several barriers to effective management. A few problems are briefly outlined in the following paragraphs, and are examined in more depth and illustrated with detailed examples in Section 3.5. Chapter 3 presents M$_b$D, an architectural framework for management applications that addresses these limitations. Chapters 4 and 5 present management applications that demonstrate how to take advantage of M$_b$D to overcome these problems. The following examples are taken from the realm of network and systems management. Note, however, that they are instances of generic problems that apply to many other types of distributed systems and applications.

Centralized Network Management does not Scale

The rigid allocation of management functions and responsibilities leads to many problems of scale. In the platform-centric paradigm, a manager process at the NOC interacts with a large number of rigid servers at the network elements. This interaction pattern allocates most processing to the platform's host computers. Thus it entails a high degree of communication exchanges that involve a single point of failure, the manager's host. This host processor establishes fundamental scale boundaries. For example, there is a limit on how many variables can be polled by a central platform and how often. Data analysis and presentation is only conducted at the central platform. Thus it requires high data access and processing rates that do not scale up for large and complex networks.

Platform-centric Network Management induces Micro-management

For most non-trivial tasks, platform management applications need to *micro-manage* the device servers. Platform managers can only interact with the device servers through general purpose interfaces. For instance, they can only invoke `Get`, `Get-Next`, or `Set` services from an SNMP-agent. Non-trivial management tasks require a large number of platform-device interactions using these generic services. Many interactions result in high communication costs and delays in responding to critical situations. Relatively large amounts of resources, such as communication bandwidth and CPU cycles, are required to accomplish even simple tasks. This high overhead can place severe restrictions on manageability, barring all but trivial tasks. A more detailed example of micromanagement is presented in Section 3.0.2.

Platform-centric Applications are Unreliable

Platform-centric management produces failure-prone communication bottlenecks. During failure times in particular, centralized management will tend to increase the rates of data access at a time when the network is least capable of handling them. Since the platform host contains most management functions, it is rendered most vulnerable to network failures. If the platform host is down or overloaded, devices cannot accomplish recovery, as they must wait for instructions from the manager. Thus, even a minor problem may potentially lead to an avalanche failure of the entire network management system, bringing it to a complete halt. Hence, the platform-centered approach is significantly limited in its ability to handle the problems that arise in complex, large-scale internets.

Platform-centric Management Imposes Resource Constraints

The type and quantity of resources available for management purposes vary greatly among networked devices. Small devices, e.g., modems, will typically offer very limited computational capabilities for management purposes. A telecommunications switch or router can typically afford much larger computing resources for

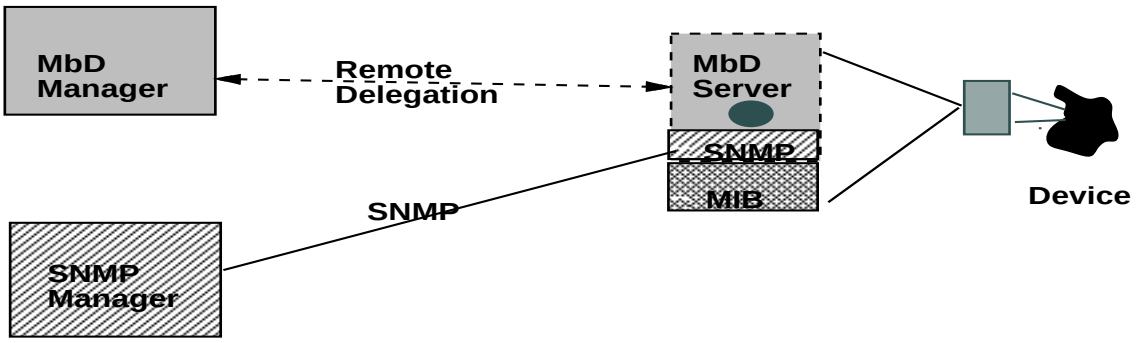

Figure 1.3: Management by Delegation

management. Some mobile devices may have limited computing resources available due to their limitations in power consumption and storage. In addition to these intrinsic limitations, administrative policies may impose additional restrictions on the allocation of management resources. For instance, a security policy may prescribe the use of strong encryption on devices depending on their geographical location. Current management standards lack any effective mechanism to differentiate between and take advantage of the capabilities of different types of devices.

1.4 Management by Delegation to Elastic Servers

Management by Delegation is the application of elastic processing to network and system management. M_bD device servers are elastic processes that implement network management functions. The approach of M_bD is to dynamically distribute the management computations to M_bD-servers at the devices where the managed resources are located. An M_bD-server is an elastic process customized for network management and provides efficient bindings to management instrumentation and support for SNMP interoperability. For instance, an M_bD-server can implement an extensible SNMP-agent, as shown in Figure 1.3. Delegated agents are sent to M_bD-servers to automate the monitoring, analysis, and control of their networked elements. For example, an operations management center can delegate management functions to an M_bD-server at a switch, programming it to execute certain tests at a given time.

Instead of bringing data from the devices to platform based applications, parts of the management applications themselves are delegated to the devices. For example, a manager host can be relieved from polling by delegating agents to the network switches to monitor and detect network failures. The switches can invoke other delegated programs to isolate and handle a specific failure. Similarly, manager applications can use delegated agents to detect and prevent intrusion attempts. Such security management functions presently require operators to manually monitor and analyze remote network accesses.

We present the design and proof-of-concept implementation of technologies to

support MbD. We compare and contrast the applicability of both centralized and distributed management paradigms for different types of management applications. Several management applications demonstrate the advantages of MbD over alternative paradigms. Chapter 4 describes *"health"* functions for compressing real time operational device data and making distributed management decisions. Chapter 5 describes external level views that dynamically extend an MIB. Implementing these functions with delegated agents overcomes most of the obstacles presented by alternative platform-centric systems.

1.4.1 MbD Advantages over Current Systems

MbD supports the dynamic distribution of network management responsibilities. Applications can use MbD to overcome many of the deficiencies of centralized network management systems. The following are some of the benefits of using MbD to implement management applications:

Dynamic Allocation of Functionality

MbD allows applications to dynamically distribute management software to the devices. Thus, it allows them to extend their management functionality as the network and its administrative requirements evolve. MbD applications can dynamically adapt to changes in the availability of network and computing resources. For instance, when an ethernet segment is overloaded during file systems backups, management applications can delegate agents that analyze data locally, and relieve the network from the polling overhead. MbD applications can take advantage of newly available resources. Networked devices are increasingly being equipped with substantial computational resources, including fast CPUs and large memories. It is cost effective to dynamically move more functionality to these less expensive devices.

Real-Time Interactions and Autonomy

MbD can dynamically control the granularity of the network interactions between the applications executing at the platform and the networked devices. Thus, MbD applications can avoid the micro-management problem. MbD applications can perform real time computations on operational data at the devices. Thus, MbD applications can react faster to device problems, and significantly reduce the network overhead and the inaccuracies due to polling. MbD-servers can improve the autonomy and reliability of many management applications. For instance, assume that a network is going to be partitioned for a few hours. A NOC application may delegate agents to execute critical management code at the devices while there is no connectivity to the central platform.

1.4.2 Development of Management Applications

The semantic heterogeneity of MIBs complicates the development of management applications. Standard management protocols like SNMP unify the *syntax* of managed data, e.g., a MIB variable can be an integer counter of ethernet frames. However, these standards do not unify the *semantics* of the managed data. The method that implements each MIB variable can be implemented in substantially different ways. MIBs allow substantial semantic variations and differences in the implementation-specific behaviors of devices. For instance, routers from different vendors are often so different that manager applications *must* use vendor-specific private MIBs to handle them.

The semantic heterogeneity of managed data complicates the development of generic management software. In the absence of such software, platform-centered management is reduced to core-dumping cryptic device data on operators' screens, i.e., *"*MIB *browsers"*. MIB browsing is not an adequate model for managing networks, as there are very few adept operators able to decipher and interpret MIB contents. Indeed, most small organizations can not afford to hire the experts required to interpret this data.

M$_b$D provides a development platform that simplifies the handling of semantic heterogeneity across devices. Delegated agents can be designed to handle the specific operational environment and distinct features of a specific device. Such agents can give the platform a higher level management view that hides many private device details. Network engineers can leverage their expertise by designing new agents that incrementally improve the overall management system. Thus, M$_b$D provides an effective environment to proactively manage networked systems of arbitrary scale and complexity.

Integration with Standards

M$_b$D-servers inter-operate with standard management protocols like SNMP. Thus, M$_b$D-servers and M$_b$D applications can be fully integrated within existing platform-centric management systems. Standard management applications can also reap many of its benefits, e.g., by accessing the computations of delegated agents via SNMP.

1.5 M$_b$D Applications

M$_b$D provides an environment for dynamically deploying network management application code. Delegating agents can help automate a larger portion of network management tasks, reducing the need for extensive human intervention in the management loop. The following sections outline two M$_b$D applications that demonstrate this capability. An extended discussion of these applications is presented in Chapters 4 and 5.

1.5.1 Compressing Management Information

Management applications need to accomplish management decisions based on observations of the network behavior. Because of the huge volume of data that characterizes the network behavior, management applications must compress real-time monitoring data at the devices. Platform-centric network management systems cannot compute many real-time management functions effectively. This is due to the *"probe-effect"* introduced by polling over the network. To enable effective decision-making, vast amounts of real-time operational data need to be compressed at their source.

Section 4.2 introduces a formal model of sample behaviors and observations of network elements. This model elucidates many of the shortcomings of current network management paradigms. In particular, we examine the observations of managed entities that are computed by SNMP MIBs, and characterize their problems and limitations. Their static approach to collecting management data wastes computing resources. Management applications that use SNMP to retrieve these values suffer from the inaccuracies that are introduced by SNMP polling.

Index Functions for Compressing Management Data

One method of compressing operational data is to compute index functions, reducing a large number of observed operational variables to a single indicator of the system state. Such indexing uses linear aggregation of a large number of variables that provide different microscopic observations. Current network management paradigms, however, do not support the temporal distribution and spatial decentralization required to compute real-time management functions effectively. Standard network management approaches require a priori knowledge of what algorithms are mapped into statically defined objects. Implementations of such observation operators must take advantage of the spatial and temporal distribution of M_bD. Management applications may use these operators to make real time decisions, e.g., to diagnose and correct element failures.

Health Functions Provide an Index of the Network State

We define *"health"* functions to provide efficient real-time compression of management data. These functions linearly combine raw MIB data into simple indexes of the network state. A health function is useful for distributed decision making. For instance, an application can program real-time reactions to a network emergency based on trends identified by a health index. M_bD supports the flexible and effective evaluation of health functions and linear threshold decisions at the data sources.

MIB objects are statically defined; that is, their algorithms are rigidly implemented in each device server. Implementing an MIB object requires prior knowledge of the corresponding algorithm. Health functions, however, cannot be included as part of a static MIB, because they may vary from site to site and over time. NOC

platform hosts can not compute them effectively, as this results in excessive polling rates that will flood the network. Centralized polling misses the original goal of maximally compressing data at its source. Such centralized computations are inaccurate because of data perturbations due to the probe-effect induced by polling.

1.5.2 Computation of MIB Views

Management applications need to compute useful information from raw data collected in MIBs. Often such computations cannot be accomplished through remote interactions between the application and SNMP-agents. For example, suppose that an application needs to perform some analysis on all the routing table entries of a router. The application can use SNMP requests to retrive the routing table entries, one row at a time. This interaction, however, does not provide an atomic snapshot of the routing table at a given time, which is needed for consistency analysis. Instead, different sections of the table as seen by the application will reflect different versions of the routing table at different times. Section 5.1 presents several additional examples of such computations showing that this model of interaction is highly inefficient and unscalable.

A central difficulty in developing management applications is the need to bridge the gap between two different data models: The application's and the MIB's. Standard network management frameworks provide no support for management applications to dynamically define external data models as part of the MIBs. Therefore, management applications are forced to retrieve large amounts of data to the platform to perform simple operations like filtering and joining MIB tables. In a multi-manager environment it is difficult to ensure atomicity or transaction semantics of management actions. Furthermore, platform-centric management does not assist applications in sharing the results of these computations with other remote managers.

Chapter 5 presents a novel technique for dynamically extending MIBs, supporting the computation of external level views. *Views* are delegated agents that allow management applications to correlate and organize collections of data which may not exist physically at the SNMP-agent. Views implement several desirable functional features which are difficult to achieve under SNMP such as data correlation, source filtering, and atomic snapshots. For example, the routing analysis application could delegate views that take atomic snapshot of MIB tables. It could then retrieve the entries of these atomic snapshots using standard SNMP `get-next` requests.

External-level management specifications are written in a special *View Definition Language* (VDL). Network engineers use VDL agents to create virtual MIB tables which contain correlated data, to generate atomic snapshots of MIB data, to establish access control mechanisms, to select data which meet a filtering condition, and to execute atomic actions.

1.6 Thesis Roadmap

We begin each chapter with a brief description of the main challenges and contributions presented in the chapter, followed by an outline of the chapter's sections.

Chapter 2 describes elastic processing with delegated agents. It motivates elastic processing via examples, and describes RDS and DBM. It discusses the performance and security characteristics of elastic processing, and compares its programming model with related work.

Chapter 3 outlines the M_bD framework and describes how it integrates with other network management systems. It characterizes the management applications that can benefit from M_bD, and describes how M_bD addresses the intrinsic problems of platform-centered network management.

Chapter 4 introduces a model to describe management observations of the behaviors of managed entities, and describes some of the problems associated with centralized polling. It shows how M_bD can be used to compress real time management data and perform management decisions.

Chapter 5 presents the MIB Computations System, which consists of a language to specify MIB view computations and M_bD service extensions that implement them.

Chapter 6 summarizes the dissertation and draws some conclusions.

Appendix A provides a brief introduction to network management, describes its functional areas and standards, and shows that it presents a broad range of non-trivial technical challenges.

2

The Elastic Processing Approach to Mobile Agents

2.1 Introduction

Current distributed computing technologies support several modes of remote interactions between executing programs or processes. Processes can access remote files using an explicit file transfer protocol (e.g., `ftp`) or implicit file server mechanisms (e.g., `NFS`); they can invoke computations at remote servers by using specialized transaction protocols or Remote Procedure Calls (RPC); they can exchange data with each other using specialized messaging protocols or distributed programming language constructs. These various modes share one feature in common: they all involve transfer of data and/or commands among *statically located* processes. Data and commands form the *mobile* parts of a computation while the programs are static.

There is a growing number of network computing scenarios which cannot be effectively addressed by such static interaction paradigms. We illustrate these scenarios through several examples.

Example: Dynamic Interaction of Clients with Web Services

Consider an information provider that wishes to use a Web server to engage users in interactive game programs. At present, these programs must execute on the server and utilize HTML pages as the presentation media to the clients. This mechanism is inadequate to engage users in fast interactions with rapidly changing graphics over relatively slow communication links. Instead, the service provider requires a mechanism to move the game programs from the server to the user's computer, link them with the local browser software environment and execute them at the user's machine. Such a transfer will be repeated as new game scenarios arise.

Example: Usage Statistics of An ATM Switch

Consider an ATM switch owned by an Internet Service Provider (ISP) that serves several thousand users in a given community. From time to time the ISP organization needs to process information about its switches, e.g., they may need switch usage statistics for capacity planning purposes. An ISP administrator will use a distributed application that retrieves data from all the switches and performs statistical analysis of the retrieved data at a central host. Let us assume that a given switch currently has some v *Virtual Circuits* (VCs), (e.g., $v = 10000$). The application will typically follow a client/server interaction paradigm that requires $O(v)$ exchanges to collect all the per-VC data of each switch. This interaction paradigm is inefficient, since it wastes network bandwidth and platform host CPU cycles.

One may claim that it will be better to predefine such processing as a service procedure of the switch, i.e., following an RPC model. However, each ISP will collect different statistics, depending on installation specific policies that evolve over time. Even if it were possible to predict all the possible services that the switch should provide, they would require the allocation of large computing resources (disk, memory) for services that are seldom used. Therefore, it is more efficient to dynamically extend the collection of services that the device provides.

Example: Monitoring News (changes) in Web Stores

Consider now a scientist who wishes to obtain reports of new information and hyperlinks posted on the World Wide Web, that relate to a specific field. At present this user must perform a distributed search for such information and retrieve all the relevant Web pages. In many cases, the user will need to compare the retrieved pages with previous versions to identify new information. This scheme is, of course, inefficient and mostly impractical. A more useful approach is to execute programs at the servers that monitor and report changes in relevant information. Such news reporting programs could be provided as a service by the server owners. However, this would require standardization of these programs to permit a user to uniformly monitor a large variety of servers. Alternatively, server owners can provide their users with the ability to execute search programs to monitor, analyze and report changes in server contents. Mechanisms are required to dispatch and execute such news monitoring programs at remote servers.

2.1.1 Mobile Agents

We use the term *mobile agent* to describe a program that is dynamically dispatched to a remote host, where it is linked and executed. This definition includes degenerate forms of agents such as postscript programs dispatched to a printer or HTML scripts dispatched to a browser. In both cases the agent programs perform limited specialized tasks. The above examples, as well as many more provided in this dissertation, clearly establish the need for more general mobile agent technologies.

The possibilities provided by mobile agents computing have recently attracted enormous interest. Indeed, several large companies[1] have described such technologies as central to their strategic vision of computing.

Challenges in Developing Mobile Agent Technologies

There are several challenges in developing effective mobile agents technologies. First, one must establish mechanisms to dynamically dispatch programs to remote hosts and link them with local resources that they must access. For example, a video game agent dispatched to a remote browser, as in the first example, must be linked with remote I/O resources and other game agents. Second, one must provide means to control the execution of these remote programs. For example, the video game may need to dispatch additional agents for different phases of the game, and control their execution following the game rules and remote interactions by other players.

Third, it is necessary to assure that mobile agents do not compromise the resources of the remote host. The Web server providers, in the second example above, need to protect access to server resources by news reporting agents. They must ensure that these agents cannot damage server pages, assume control of OS resources, or utilize excessive CPU cycles and/or memory to perform their functions. Fourth, it is necessary to integrate mobile agent technologies within current heterogeneous distributed computing mechanisms and environments. For instance, the video game players may have very different personal computers, in terms of type and amount of available hardware (CPU, memory) and software (different operating systems).

2.1.2 Language-Based vs Process-Based Agents

Several proposals for mobile agent technologies have been recently described by various groups including Java [Gosling and McGilton, 1995], Telescript [White, 1994], and Safe-TCL [Borenstein, 1994]. These proposals share a fundamental common base: agents are program scripts written in the corresponding language that are dispatched to a remote interpreter where they are executed. For example, Java scripts are retrieved by a HotJava browser [Gosling, 1995] and executed by the Java interpreter at the browser. One can script games and various other local interaction loops in Java and provide these scripts as extensions of standard Web information services. Java thus enables remote dynamic extensibility of Web browsers.

This thesis introduces a completely different approach to mobile agents computing, namely *elastic processing*. Elastic processing consists of two components:

- a Remote Delegation Service (RDS) to dispatch an agent to a remote elastic process, invoke it as a thread of the process, and control its execution.
- An elastic process structure to support dynamic delegation, linking and remote control of agents.

[1] including Oracle, Sun and IBM

For example, a video server provider can use RDS to delegate an agent to interact and play a game with the user and other remote players. Depending on the rules of the game, additional agents are delegated, linked, and executed as threads of an elastic process that provides controlled access to the user's personal computer. Similarly, a news reporting agent can be dispatched, using RDS, to remote Web sites to monitor their server contents, detect changes and report them. The agent program will be downloaded to the servers and executed whenever respective pages are modified or new pages are added.

Delegated agents and elastic processing offer several advantages over language-based scripting agent technologies. First, agents can be developed in compiled or interpreted languages using current programming languages and development tools. Compiled agents written in industrial-strength languages, e.g., C or C++, can be used to perform real-time tasks or to construct complex systems well beyond what is possible using agents scripted in an interpreted language. For example, one can delegate an agent to a remote TV decoder to decompress and display a video stream that uses a novel compression technology. Similarly, one can delegate agents that support new protocol interfaces at a remote system and thereby extend its range of interoperability.

Second, one can delegate an entire interpreter as an agent and then delegate scripts to it. For example, one can delegate a TCL [Ousterhout, 1990] interpreter to an information appliance, and then delegate and execute TCL script agents to present information to users in interactive forms not supported by the Web. When the interactions are completed the interpreter and script agents may be discarded, saving resources for other types of computations. Thus, elastic processing provides support for language-based scripting agents as a special case.

Third, elastic processes can extend standard O/S access control mechanisms and apply them to accomplish secure agents. For instance, a delegated agent may execute under a predefined userid (e.g., `nobody`) with very limited access to local resources. Such a mechanism will allow a delegated game to interact with the PC's screen and audio device, but will prevent it from accessing personal files of the user. Also, elastic processes may enforce authentication mechanisms which ensure that only agents from reputable sources are accepted. For example, the games server may only accept agents that have been *fingerprinted* by the service provider.

Fourth, RDS provides extensive execution control capabilities that permit flexible models of agent control. For example, a delegated game agent may be dispatched to the client host and then invoked periodically under local or remote scheduling control. Depending on the game interactions, the agent can cause the delegation of additional agents to the user's host and to the hosts of other players to pursue different game plans. The execution of these agents may be controlled by the remote server in order to reflect the game interactions of all the players.

Chapter Organization

- Section 2.2 presents a few examples of non-elastic processes that illustrate the problems that elastic processing addresses. It formally defines elastic processes, and introduces RDS.
- Section 2.3 presents the architecture of the Delegation Backplane Middleware that implements the runtime of elastic processes.
- Section 2.4 outlines the security requirements of RDS and presents a security model for elastic processing.
- Section 2.5 analyzes the performance characteristics and tradeoffs of elastic processing via a simple application example.
- Section 2.6 presents several examples of distributed applications that can benefit from elastic processing, and discusses the common characteristics of these applications.
- Section 2.7 compares elasticity with other related work, including RPC, Remote Execution, Remote Evaluation, and Remote Scripting.
- Section 2.8 presents conclusions.

2.2 What is Elastic Processing?

Consider again the ATM switch example presented in Section 2.1. The application engaged in a client-server interaction that was inefficient, since it wasted network bandwidth and platform host CPU cycles. An alternative approach is to distribute the data processing computations to where the data is located. In this example, the application could delegate the computation of the statistics to the switch itself, thereby avoiding network delays.

Elastic Processing Supports Spatial and Temporal Distribution

The above example is representative of a large collection of emerging networked applications that can take advantage of dynamic extensibility. *Elastic Processing* is a novel distributed computing paradigm that supports dynamic extensibility of remote software processes, i.e., it is both *"spatial"* and *"temporal"* distribution. Elastic processes are executing programs that can dynamically integrate new functionality sent to them from external processes as *delegated agents*. Elastic processing is language independent and supports explicit remote control of agent's execution. The technologies that support remote elasticity consist of a *Remote Delegation Service* (RDS) and a multithreaded *"Delegation Backplane Middleware"* (DBM) architecture for elastic processes.

A delegated *agent* is a program code that is (1) dispatched to an elastic process host, (2) translated and dynamically loaded, (3) executed as a thread inside the address space of the elastic process, or as an independent process in the same host, (4) controlled remotely by authorized parties. Thus, an external (local or remote) process can dynamically extend the functionality of an elastic process, by delegating an agent to it. Dispatching and executing an agent are two decoupled asynchronous actions, that can be initiated by the sender or by the elastic process receiver.

RDS provides applications the ability to (1) remotely configure an elastic process, (2) control the execution of delegated agents, and (3) convey information to and from these agents. The DBM runtime environment implements a software *"backplane"* where delegated programs are loaded and can execute as threads in a shared address space. DBM supports translation and dynamic linking of delegated code, a multithreaded execution environment, dynamic resource allocation, and interprocess communications.

Elastic Processing Requirements

In this chapter we evaluate the requirements of distributed applications and show the following:

- Network delays are increasingly becoming a major performance bottleneck for many types of distributed applications. Elastic processing provides an effective paradigm to overcome such network delays.
- Remote dynamic extensibility requires a security model that prevents its abuse by unauthorized parties. We define a language-independent, configurable security model based on party authentication and controlled agent execution.
- Application programmers need to customize their distributed applications to meet changing requirements. We present a collection of application examples that can benefit from elasticity, and outline their main characteristics.

2.2.1 Why Do We Need Elastic Processes?

This section presents two brief examples that illustrate some of the problems addressed by elastic processing.

A Web Browser Process

NCSA Mosaic [2] is a hypertext browser program that implements a World Wide Web rigid client. It has hard-wired knowledge about many different file formats (e.g., gif, html) and protocols (e.g., http, smtp, ftp). The browser uses this knowledge to access remote information which is available at servers that provide data in an appropriate format and support a corresponding protocol. The browser can be statically

[2] See `http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html`

configured to invoke different programs for each data type, as defined in a special mapping configuration file. For instance, it is possible to configure the browser to choose the `xdvi` program as the browser for all `.dvi` objects. This startup configuration provides a limited degree of flexibility in the application's behavior, but it does not modify the browser process itself.

A specific implementation of the Mosaic browser can not handle protocols that have not been rigidly defined within it. For example, Mosaic can not handle a *Universal Resource Locator* (URL) like `mynewprot://foreign.edu/obj1.mnp`, if the `mynewprot` protocol has not been predefined. In other words, the browser can only access objects whose URL protocol is predefined, e.g., http, smtp, ftp. To support a new protocol, say S-http, the implementation of the browser must be extended. If the browser could be dynamically extended, new protocols could be added as needed. HotJava [Gosling, 1995] is an example of an extensible client browser. We compare their model with ours in Section 2.7.4.

An Rmtd Daemon Server Process

`rmtd` [Uni, 1986; Stevens, 1990] is a Unix "deamon" server that provides remote file access over magnetic tape drives. The `rmtd` process supports the following service requests: `open, close, lseek, write, read, ioctl,` and `status`. For example, a successful `status` request returns to the client a *binary* buffer which contains hardware-dependent information describing the current status of a tape drive.

`rmtd` clients are limited to selecting choices from a predefined set of services. In order to customize distributed applications, program designers often have to add significant functionality to the client processes. For example, an `rmt` client needs to convert and interpret the binary data returned by a `status` request, which varies for each tape drive. This static allocation of functionality leads to a non-modular and unscalable design. Each client process has to support multiple routines to compensate for the lack of appropriate server functionality, in this case, to interpret the device-dependent binary data. Shifting functionality to the clients complicates the development, deployment and maintenance of distributed applications, as there are typically many clients per server.

A tape drive device vendor could provide extensions to the server to support additional functionality. For example, the designer of an alternative `rmtd`-like program could add an additional service entry-point, e.g., `status_text`. This additional service could return a self explanatory *ASCII* string. Such extensions will have to be compiled and linked with the rest of the original server code.

Executing clients may not want to be affected by such server changes. Particularly, they may not want to become aware of the server upgrade by losing connections with it or by other exceptional conditions. Upgrading a rigid server is likely to entail installing a private copy of it. Keeping multiple copies of the server code requires additional storage space. Then it becomes difficult to upgrade, maintain, and manage each version of such program. The following section outlines some of the generic problems of distributed computing with rigid processes.

2.2.2 Formal Definition

This section formally defines elastic processing. An elastic process is a multithreaded process that supports the remote invocation of a set of elastic transformations. These transformations allow remote processes to (1) extend the functionality of the elastic process by delegating agents to it, (2) control the execution of the agents, and (3) communicate with the agents.

A process $\Pi \equiv< C, S >$, consists of a program code C and a state S. For example, a file editor process that I am currently using is an executing incarnation of the `emacs` program (C). The state (S) of this process includes the current contents of several files, personal customizations, and so forth. C is defined by a collection of code program fragments that Π can execute, $C \equiv \{c_1, c_2, ..., c_n\}$. For my `emacs` process, the c_i include all the code in the main program executable, and all the imported library routines that were explicitly and implicitly loaded into its address space. The process state S is defined by the state of all its threads, $S \equiv \{s_1, s_2, ..., s_m\}$, where $s_i =< c^i, x^i >$. c^i indicates the code associated with each thread, and x^i its execution state.

Note well that $|C| = n \neq |S| = m$, i.e., the number of threads is independent of the number of program fragments. Consider, for instance, a multithreaded appointment server process, where a new thread is allocated for each appointment. The state of each thread includes the program code that the thread executes for some type of appointment transaction and the data associated with it. There may be several appointment threads that share the same code but have different states.

Elastic Transformations

An elastic process Π_E is defined by two *dynamic* sets C and S, which can be modified via elastic transformations. An elastic process supports the remote invocation of the following code extensibility and state control transformations.

Code Extensibility Transformations

Code Extensibility Transformations allow a remote process to extend or contract the program code set, C, of an elastic process.

- An *addition* transformation incorporates a new code fragment, c into an executing process, $\Pi =< C, S >$.
  $Addition : \{< C, S >, c\} \mapsto< \{C \cup c\}, S >$

- A *deletion* transformation deletes a code fragment c from a process P, and may implicitly remove any thread whose state is associated with the deleted code fragment.
  $Deletion : \{< C, S >, c\} \mapsto< \{C - c\}, \{S - \{s_i : s_i =< c, x^i >\}\} >$

State Control Transformations

State Control Transformations allow a remote process to modify the state of an elastic process.

- An *instantiation* incorporates a new thread to the state of an existing process.
  $Instantiation : \{< C, S >, c\} \mapsto < C, \{\{s_1, ..., s_k\} \cup \{s_{k+1} =< c, Run >\}\} >$
- A *termination* removes a thread from an existing process.
  $Termination : \{< C, S >, s_j\} \mapsto < C, S - s_j >$
- A *suspension* changes the state of an executing thread to suspended.
  $Suspension : \{s_j =< c, \_ >\} \mapsto \{s_j =< c, Sus >\}$
- A *resumption* changes the state of a suspended thread to executing.
  $Resumption : \{s_j =< c, Sus >\} \mapsto \{s_j =< c, Run >\}$

Elastic Processes and Elastic Servers

An *elastic process*, Π_E, supports the remote invocation of the above six elastic transformations. That is, the code and state of a Π_E can be remotely modified, during its execution, as a result of an explicit external interaction. An elastic process is a generalization of dynamic linking to a distributed, multithreaded environment. The Remote Delegation Service (RDS), implements the elastic transformations, and communications between process threads.

An *elastic server* is an elastic process whose service interfaces can be remotely modified. That is, the interface of the server is a dynamically changing collection of service procedures, $\{p_1, p_2, \cdots, p_l\}$ that can be remotely invoked by its clients. New p_i procedures can be added and existing procedures can be removed via remote extensibility transformations.

2.2.3 Remote Delegation Service

RDS enables processes to exchange agents and to control their execution. A delegator process uses RDS to transfer a *Delegated Program (DP)* to an elastic process and to control its execution. An elastic process compiles and dynamically links a delegated agent, and returns a handle to the delegator. DPs are instantiated as independent threads (DPIs) in the address space of the elastic process. Instantiated agents can establish bindings to exchange messages or to make remote calls. Elastic processes implement the RDS interfaces listed in figure 2.1. RDS enables remote processes to:

- Transfer a delegated agent to an elastic process.
- Instantiate a delegated agent as a lightweight thread or full process.
- Suspend, resume, or terminate the execution of a delegated agent.

```
RDS_Delegate(EP, DP, &DPid);
RDS_Delete(DPid);
RDS_Instantiate(DPid, &DPIid);
RDS_Terminate(DPIid);
RDS_Suspend(DPIid);
RDS_Resume(DPIid);
RDS_SendMsg(DPIid, Msg);
RDS_ReceiveMsg(&Msg);
```

Figure 2.1: RDS Services

- Exchange messages with the delegated agents.

Figure 2.2 illustrates the use of these operations. The remote process *delegator* is assumed to be authorized to perform the corresponding actions:

1. A DP is being delegated, using RDS_Delegate(EP, DP, &DPid), to an elastic process EP.
2. The DP is compiled and dynamically linked with the elastic process address space.
3. The delegator process executes RDS_Instantiate to create a DPI.
4. The delegator (and/or other processes) communicates with the DPIs using RDS_SendMsg and RDS_ReceiveMsg.
5. The delegator process can suspend and resume the execution of the DPI, using RDS_Suspend and RDS_Resume.
6. The delegator aborts the execution of the DPI, using RDS_Terminate.
7. Finally, the delegator removes the DP using RDS_Delete.

Note that the entity invoking a RDS service may be either remote or local (another process in the same host, or another thread in the same process). We now examine each of these services in more detail.

Delegation and Deletion of Agents

Using RDS_*Delegate*, a delegator process (Π_D) requests that a delegated program (DP) be incorporated to the elastic process Π_ε. This action performs the actual transfer of the program code, e.g., using the trivial file transfer protocol `tftp` [Sollins, 1992]. For an RDS_*Delegate* to succeed, the following actions must be completed by the underlying runtime support of Π_ε:

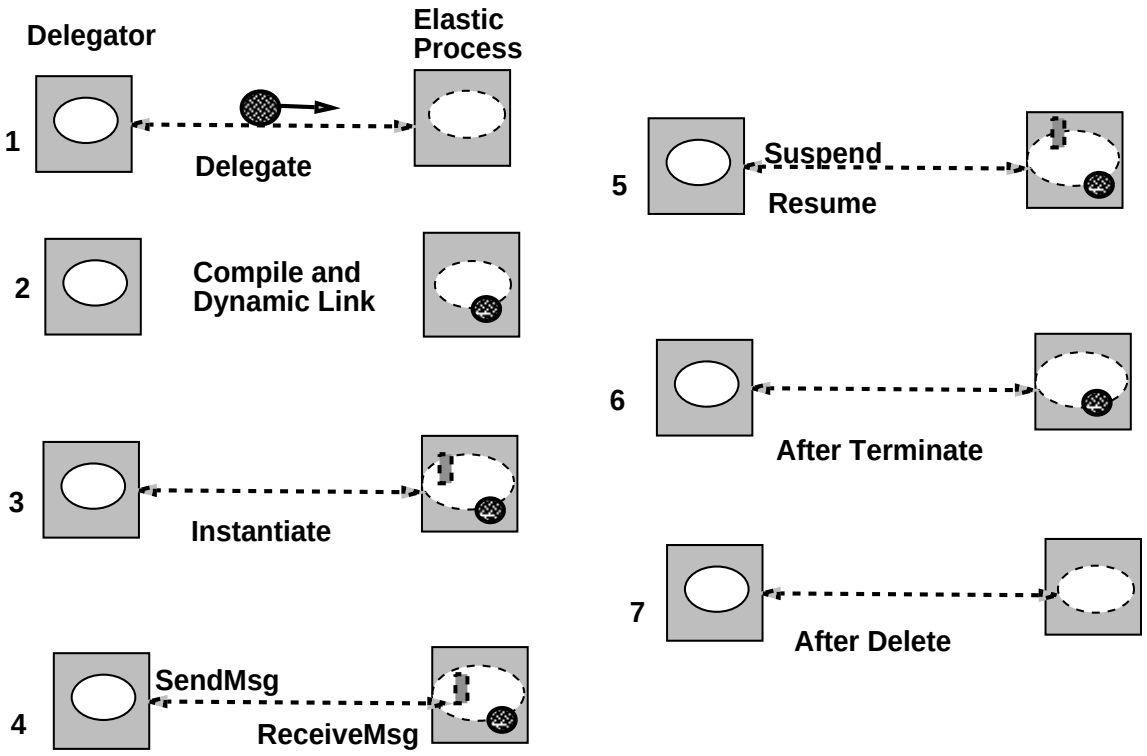

Figure 2.2: Using RDS Services

- The delegating process must be authenticated according to a predefined security policy.
- The DP must be checked by a translator to ensure that it is a legal program that complies with its predefined language rules.
- The DP must also fit under the current resources available in Π_ε.

These actions are all performed implicitly by the underlying elastic process runtime, without any explicit application intervention. When a DP is successfully delegated, the elastic process runtime returns to the delegator a DP *handle* (`DPid`). This handle identifies the DP and is later used to instantiate or delete it. The delegator process can share a DP handle with other processes, allowing them to create additional instances of the DP.

A remote process can cause the deletion of a DP previously delegated to an elastic process. This action performs the actual unlinking and deletion of the program code of DP at the elastic process, as described in Section 2.3.2. For an RDS_*Delete* to succeed, the following actions must be completed by the underlying runtime support of Π_ε:

- The deleting process must be authenticated as one with the appropriate right to delete the DP.
- The `DPid` must refer to a valid DP.
- All executing DPIs which are instances of the DP being deleted are terminated.

Scheduling Control of DPIs

Scheduling control allows remote processes to instantiate, suspend, resume, and terminate DPIs. For any one of these operations to succeed, the Π_ε runtime authenticates the remote process to ascertain that it has the appropriate permissions. The state of an elastic process includes two sets of threads: active and suspended. An instantiated thread is in the active set until it is suspended. A suspended thread is not scheduled until it returns to the active set through a resume action.

The RDS_*Instantiate* service allows processes to remotely instantiate a new thread DPI from a previously delegated DP inside the address space of an elastic process. When a DP is instantiated, the creator receives a `(DPIid)` handle that identifies the DPI and enables control operations over it. The RDS_*Terminate* service terminates a DPI. RDS allows remote processes to exercise scheduling control to suspend and resume the execution of a DPI. RDS_*Suspend* moves a DPI to the suspended set, and an RDS_*Resume* restores it to the active set.

Message Communications with DPIs

RDS supports communications by message exchange between remote processes and DPIs. A remote process can send a message M to a DPI d using RDS_$SendMsg(d, M)$. The RDS communication support locates the elastic process and delivers the message to its runtime, which forwards it to the corresponding DPI. It is up to the DPI to interpret its contents and act accordingly. This operation is similar to sending a datagram. RDS_$ReceiveMsg(\&M)$ allows a DPI to receive a message that was sent using RDS_*SendMsg*. DPIs can use the same service interface to communicate amongst themselves.

2.2.4 A Sample Scenario

Let us illustrate a sample application of RDS. Consider a *Travel Assistant Application*, "TAA", that executes on a mobile laptop computer of a travelling businessperson. For example, the TAA application may assist its user in negotiating settlements to claims. The application will access information from servers located at the home office, to retrieve old e-mail messages, summaries of previous discussions, and so forth. Let us assume that the laptop computer is connected to the home server via a low-bandwidth modem link, e.g., 4800 bps. The user of the TAA applications obviously wants to minimize the cost of the phone call, and therefore TAA needs

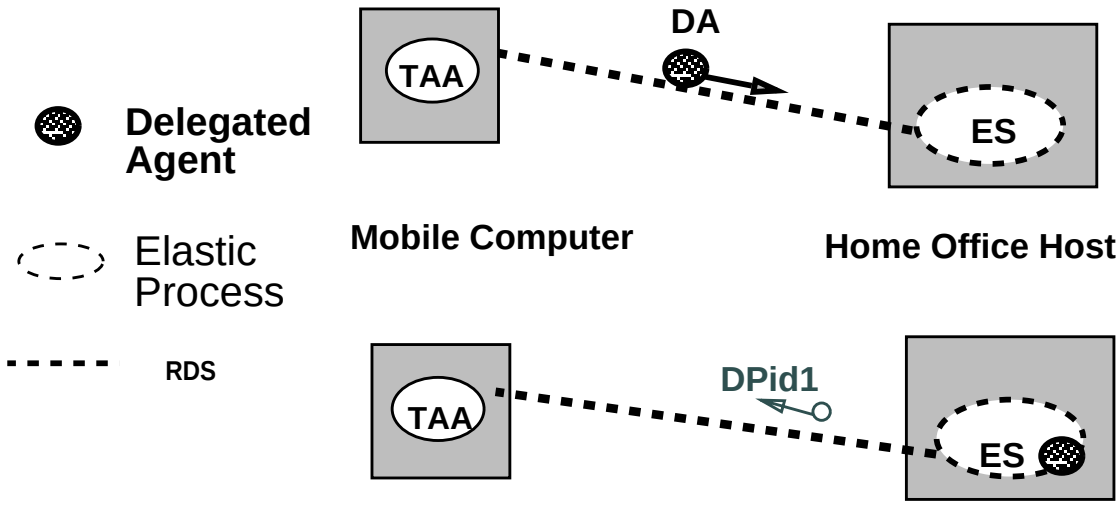

Figure 2.3: Delegating an Agent to an Elastic Server

to retrieve the required information quickly. The TAA application will use RDS to minimize the data exchanges required to retrieve the pertinent data.

Suppose that the traveler needs to retrieve a collection of old e-mail messages that pertain to a set of related claims. These messages are stored at the home office on tape. The TAA front-end will first ask the user to provide some pertinent data for selecting the e-mail messages for retrieval. For instance, the user may indicate date ranges, a set of senders and receivers, and specific keywords. The TAA will then compose an agent, DA, which combines the information provided by the traveler and agent code templates from its library.

A snapshot of this delegation scenario is shown in Figure 2.3. The TAA client process has established an authenticated and secure connection with the elastic server ES. In the top part of the figure the TAA is performing a `RDS_Delegate(ES, DA, &DPid)` operation. This action delegates DA to the elastic server, ES, at the home office host.

DA can be accepted or rejected by ES, due to security or safety considerations. For instance, DA may be rejected on security grounds because the TAA is not authorized to delegate agents to the ES server that controls the tape archive. DA may be rejected on safety grounds because its program tries to invoke a service that violates a safety rule. For instance, ES may have a safety rule that prevents clients connected via modem to `write()` to archival tape files. If DA is accepted, it becomes an integral part of ES, and the TAA then receives a handle to it. On the bottom part of the figure, ES has completed the integration of the delegated agent and is returning to the TAA the `DPid1` identifier of the delegated agent. DA will execute inside ES the required operations, e.g., `read()` from the corresponding tape.

Using `DPid1`, the TAA can instantiate DPI threads, d_i, to execute the code of

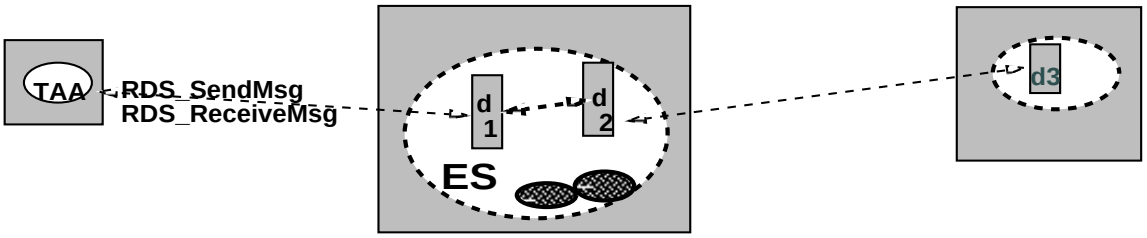

Figure 2.4: Communicating DPIs

DA inside the address space of ES. For instance, the TAA may instantiate a number of DPIs, each performing the same search concurrently on different tapes. To create each DPI, the TAA invokes RDS_`Instantiate(DPid1, &DPIid)`. If the instantiation is successful, a new thread d_1 is activated and it starts executing the code of DA.

TAA receives a `DPIid` handle to d_1, which can be used to control its execution. For example, the TAA may need to suspend the execution of d_i for some time, when it notices a local event that requires its full attention. For instance, suppose that in the middle of executing the TAA application, an urgent fax for the user is received at the server. The TAA application may notice this, and will temporarily suspend d_1 in order to free bandwidth to speed up the reception of the more urgent fax at the mobile laptop host. After the fax was received, the TAA application can resume the execution of d_1.

The DPI d_1 can communicate with external processes and with other DPIs, as shown in Figure 2.4. In the above example, d_1 may send the results of its e-mail searches to another DPI, d_2 which performs sorting, encryption and compression on its output, before returning the results to the mobile host. Furthermore, d_1 could communicate with DPI executing on a different elastic server, e.g., d_3. For instance, d_3 could provide addressing information needed for the queries. When the appropriate application information has been retrieved, the TAA can terminate its DPIs and delete its DPs using the corresponding RDS services.

2.2.5 Benefits of Elastic Processing

Resource Consumption

C/S applications often make implicit requirements for resource-rich, powerful computing environments. Many popular distributed applications, like `Mosaic`, are optimized to execute on high-performance workstations and communicate over high bandwidth links. Distributed applications which are popular in resource-rich workstations, are often rendered almost useless in resource-constrained environments, such as mobile devices. Such devices are intermittently connected to wireless networks, and have low-bandwidth connections, particularly outside office buildings. Mobile devices have limited computing power, in part due to battery limitations.

Figure 2.5: An elastic Web Proxy

Example Web Browser on Small Mobile Device

Consider a few of the problems of implementing a Web browser on a small mobile device, e.g., a palmtop device like the HP200, which offers a tiny screen and approximately 2MB of storage.

- Since HTML pages are typically designed for larger displays, users of this browser will need to scroll frequently.
- Because of the limited storage, the device cannot keep large graphic files files retrieved via HTTP.
- Because mobile communications are typically performed over low-bandwidth, error-prone wireless links, they can significantly hamper and delay remote data exchanges. Following a URL hyperlink assumes that there is an available connection to the Web server, but these connections are often unavailable.

Flexibility in the allocation of processing tasks may enable applications to execute appropriately in environments of different resource characteristics. For instance, the palmtop device could use an elastic process on a stationary workstation to act as a proxy HTTP client. Such a proxy would execute most data retrievals over a high bandwidth connection. Agents could be delegated from the palmtop device to the proxy to filter, reformat, and compress Web pages, reducing their bandwidth and screen requirements. This is illustrated in Figure 2.5.

The elastic server proxy will perform most of the data exchanges with remote Web servers, thereby reducing the delays due to the wireless connection. The per-minute usage cost of remote interactions over low bandwidth wireless connections may be very high. Adapting application processes to incorporate compression algorithms as needed decreases their bandwidth requirements, and hence reduces their wireless phone bills. The laptop itself could be augmented by the proxy with customized code to decompress each Web page. The laptop browser could also maintain a cache of frequently used (or soon to be used) Web pages. The proxy can track the content of this cache and refresh it when the original Web pages change.

Performance of Interactions

For many distributed applications, dynamic process extensibility can improve interaction performance. For instance, elasticity may reduce the number of process-to-process interactions. For instance, the delay experienced by a Web browser when retrieving remote data varies significantly depending on its network connection. Mobile computers, in particular, are attached to different network connections at different times. For instance, they can be attached to a docking station with a reliable 10^8 *bits-per-second* (bps) LAN, or with an unreliable wireless connection at 10^3 bps.

The application's performance could be improved significantly if the client process adapts different protocols for use depending on the available resources. For instance, in the above example this could be done by retrieving only text from the proxy to the laptop when using a low-bandwidth connection link. On the fly extensibility of a process can assist in adapting the application to dynamically changing environments. Several such problems are illustrated with examples of distributed management applications in Section 3.5.

Program Size

Mobile devices have limited storage capacity, due to weight and price considerations. While there are many expensive laptops with large resources, there are many small devices which only offer very limited amounts of storage. Elasticity can reduce the size of the application code that must be present at the device. Many client programs require large disk space for their code and for their temporary data. Some versions of `Mosaic`, for instance, require almost 1MB of storage for the executable image. Other useful clients are even larger, e.g., a fully configured `Notes` client requires more than 50MB. Large size clients require many diskettes for distribution, making it difficult and expensive to give away free samples. Large size makes it very difficult to transfer an entire client to a remote host, particularly over low bandwidth connections, e.g., phone lines.

Implementing an application component as an elastic process can facilitate its distribution. One need only distribute a small core of the process, and it will dynamically retrieve additional functions as needed. Distributed applications can store temporary data at intermediate proxy servers and use delegated agents to page them. For example, in the Web browser example above, the space required for caching large image files (e.g., `.gif` files), can be allocated at the proxy server host.

2.3 Architecture of Elastic Processes

The software architecture of the elastic processing runtime environment is stratified in three layers, as shown in Figure 2.6.

- At the top is the *application* layer, where DPIs execute as threads that exchange messages via an asynchronous communication service.

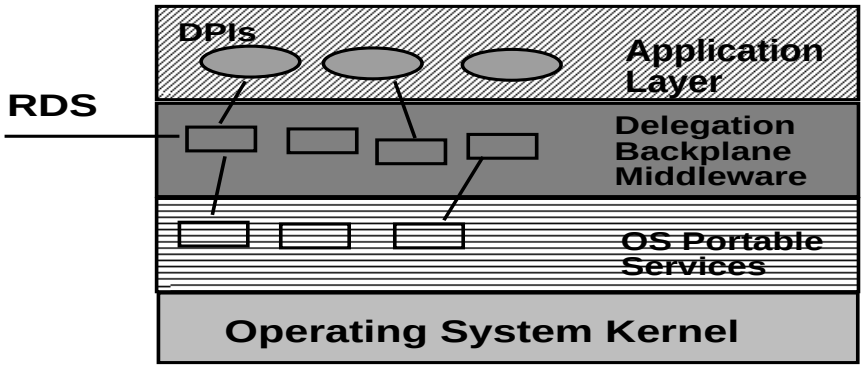

Figure 2.6: Elastic Process Runtime Layers

- The second layer implements a *"Delegation Backplane Middleware"*, DBM, that provides support for RDS operations. DBM allows the integration of DPs within an elastic process, the instantiation and execution control of DPIs, and communication between them. Like an operating system microkernel, DBM provides a small set of services.
- The third layer is a "thin veneer" that provides an API for accessing O/S services. The second layer implementation assumes that the underlying O/S services have POSIX [Lewine, 1991] semantics.

This section describes how the DBM components support RDS. RDS allows processes to add, modify, and delete the services of an elastic process without stopping its execution. To provide this functionality, DBM must:

- Ensure that DPs follow some set of rules.
- Appropriately store and integrate DPs within the elastic process.
- Establish a flexible security policy for elastic processes, e.g., authentication.
- Support the RDS remote control of DPIs.
- Support communication between DPIs and the external environment.
- Manage the allocation of resources between the DPIs, protect concurrent accesses to shared objects, and handle all asynchronous events.

DBM is implemented by a collection of interacting thread components, each responsible for certain aspects of the above issues. Figure 2.7 depicts the internal structure of the DBM and the relationships between its component threads: the *Controller, Protocol, Repository, Translator,* DPI-*Manager, IPC, and Scheduler.*

The Controller executes as the initialization thread to configure the elastic process DBM. When a delegator process performs a RDS_Delegate, the Protocol component receives the DP. The Repository stores the DP, and assigns it a unique external

identifier, the `DPid`. The Translator compiles and links the DP. Requests to instantiate DPs are forwarded to the DPI-Manager. The DPI-Manager implicitly interacts with the Scheduler to support the RDS_Suspend, RDS_Resume, and RDS_Terminate requests. The Scheduler implements a preemptive multi-priority round-robin policy. DPIs communicate by invoking services provided by the IPC component. The following subsections describe in more detail the functionality of each component.

2.3.1 Controller

The *Controller* initializes the DBM runtime environment by instantiating all the other component threads. It reads configuration instructions from an initialization file, e.g., `.dbminit`. Policies which are invariant for the lifetime of the elastic process are implemented as DPs which are instantiated by the controller. A typical line in this file is an instruction to load and initialize a new *load-time* thread for a specific function. Load-time threads can be either DBM *configuration* threads or *application specific* threads which are being pre-delegated. The controller can be instructed to self-delegate a DBM configuration thread to support a specific authentication scheme. An example of an application specific DPI is an SNMP-agent thread that supports MIB requests. Such an SNMP-agent is application specific but requires load time initialization of its data structures.

2.3.2 Protocol

This component implements the protocols that allow remote processes to request RDS services and to communicate with DPIs. It implements a file transfer protocol to delegate DPs, and exchanges messages with remote processes to support the RDS services. The Protocol thread becomes a proxy for remote processes, locally invoking RDS services on their behalf. For instance, if a DPI sends a message to a remote process, the Protocol thread forwards it to its destination.

The first prototype implementation supported only a trivial access control authentication for DPs and DPIs based on their respective handles. A later implementation [Dupuy, 1995] added optional MD5 [Rivest, 1992] authentication (see Section 2.4). The protocol component is designed to support different network protocols and data representation encodings. The current prototype is implemented over the BSD socket interface and uses either TCP connections or UDP datagrams of the TCP/IP protocol suite [Comer and Stevens, 1993]. It uses the ASN.1 Basic Encoding Rules [ISO, 1990b] to encode RDS message headers.

2.3.3 DP management

Repository

The *Repository* provides a common database service to store DPs in the underlying file system. This interface allows it to store, lookup, and delete DPs. For

example, after receiving an RDS_Delegate request, the Protocol module stores the intermediate file with the Repository. If the delegation attempt is successful, the Repository returns a handle to the DP, `DPid`. The Repository provides a naming service to map between these `DPid`s and the internal names used by the file system. The Repository maintains an internal data structure representing each DP. This structure contains a file system reference to the actual code and additional information describing DP characteristics, such as usage count and authentication related information.

Translator

The Translator compiles source code DPs, and stores the output object code in the Repository. If the DP violates any of a set of defined rules for the given language, the DP is rejected. For example, a prototype elastic process supports a specific subset of the ANSI C [ANSI, 1989] standard as the base language for encoding DPs. This subset language restricts DPs on their ability to bind to external functions. The DBM runtime maintains a predefined set of *allowed* functions. If a DP invokes an arbitrary external function which is not listed in this list then the DP is rejected. A language compliant DP is dynamically link-edited into the address space of the executing elastic process. DP languages are further discussed in Section 2.3.6.

2.3.4 Thread Management

DBM support for DPI threads is provided by the DPI-Manager, IPC, and Scheduler components.

DPI-Manager

The DPI-Manager allows remote processes to instantiate, kill, suspend, and resume DPIs. When a process (local or remote) wants to instantiate a DP, the corresponding message is forwarded to the DPI-Manager. This component also implicitly interacts with the Scheduler to support the scheduling requests RDS_Suspend, RDS_Resume, and RDS_Terminate. The DBM runtime protects concurrent accesses to critical regions, preventing threads from concurrently accessing shared data structures. For example, DPIs may dynamically allocate memory, and this memory must be managed by the runtime environment. The runtime provides a "jacket" for the memory allocation library routine, i.e., `malloc`, to maintain bookkeeping of the allocated memory per-DPI. When a DPI terminates, its resources (memory, implicit locks, etc.) are freed.

IPC

The IPC component supports efficient asynchronous communications between DPIs on the same elastic process over shared memory. It provides a send/receive interface that masks all the complexity of handling shared memory. Messages are

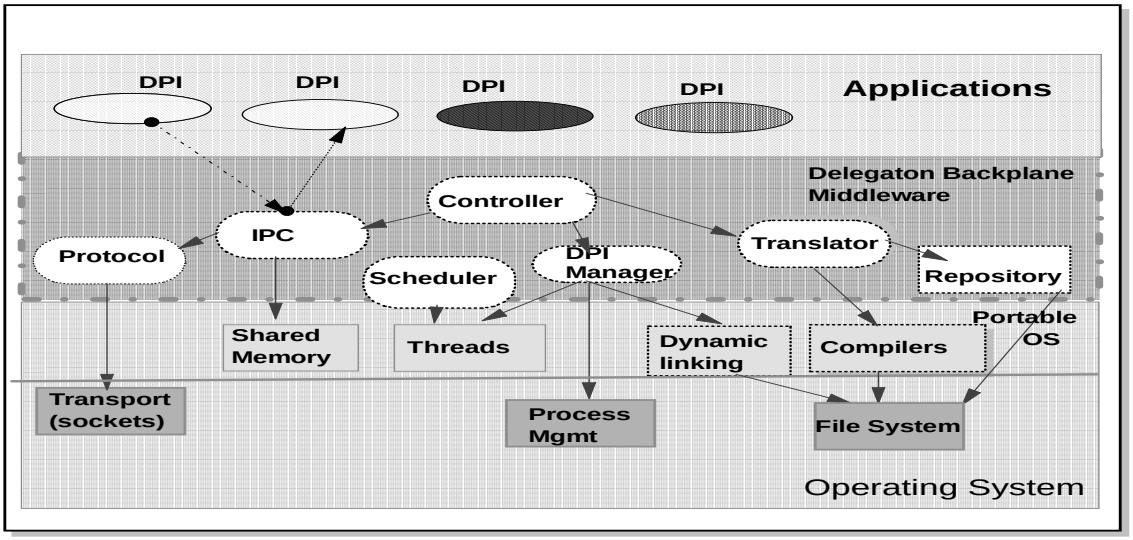

Figure 2.7: Elastic Process Runtime Environment

sent and received in the same way for both local and remote partners. DPIs can either poll or wait to receive a message. It also provides shared access to I/O facilities, and traps and handles all asynchronous events and signals.

Scheduler

The Scheduler provides a programmable, preemptive multitasking scheduling facility. A DPI thread is put to sleep either when it waits upon some resource, runs for the full duration of its time quantum, or voluntarily relinquishes the processor. A DPI yields the CPU implicitly by waiting for some event or explicitly calling a function to put itself to sleep. The entire state of the threads is maintained in user space, i.e., no O/S kernel resources are allocated per thread. Thus, DPI switching is efficiently done without changing address space. Section 3.3.4 presents scheduling mechanisms to implement DPI's priorities.

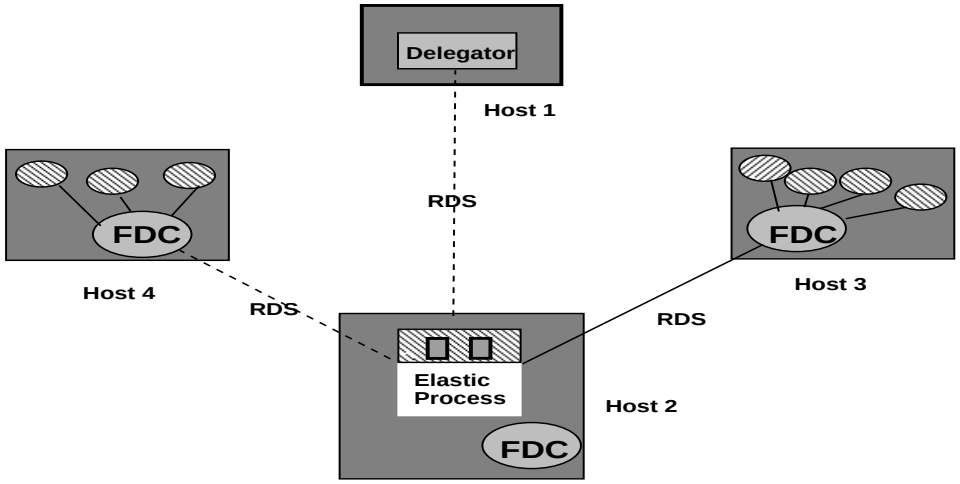

Figure 2.8: Full Process DPI Controllers

2.3.5 Full Process DPIs

The *Full-Process* DPI *Controller* (FDC) process supports the execution of DPIs as full address-space processes. The FDC supports RDS requests to control heavyweight DPIs, i.e., instantiate, suspend, resume and terminate them. The FDC is a distributed extension of the DBM, which acts as a demon process at each host. The FDC communicates with the elastic process DBM, and controls process instances executing in its host. Messages sent to and from a heavyweight DPI are forwarded by the Protocol thread to the corresponding FDC. Figure 2.8 shows FDCs on hosts 2, 3, and 4. Each FDC controls the execution of the full-process DPIs in its host. Remote processes from any host can communicate with any DPI, and the corresponding FDC will forward their messages.

Comparison of Full Process vs Thread DPIs

Supporting DPIs as full (e.g., Unix "heavyweight" address space) processes has several advantages. A full process can be easier to port across heterogeneous environments, and can provide greater flexibility of resource allocation. If the DPIs are computationally bound, distributing them among several hosts provides an effective way to harness additional computing power. For example, a distributed geology application used full process agents written in Concert/C [Auerbach *et al.*, 1994a] to collect and analyze seismic sensored phenomena [Soares and Karben, 1993].

A multithreaded elastic process presents a single unit for operating system enforced resource constraints. For instance, a single elastic process is limited in the number of concurrently open files that can be shared by its DPIs. If there is a need for more resources per DPI than available, we can either create more instances of elastic process or execute the DPIs as full processes with a larger allocation of

resources. Also, DPIs that may be unreliable or untrusted can be tested in a more secure, encapsulated environment without introducing risks to the rest of the elastic process. For instance, an elastic process may receive a delegated agent from a suspect source, e.g., an anonymous posting. To prevent harm to its internal resources, the elastic process may execute such an agent as a separate process, thereby using existing O/S mechanisms to protect processes.

Implementing DPIs in a multithreaded environment offers many advantages:

- The context switching time for a thread is typically much lower than for a 'heavyweight' processes.
- DPI threads may execute in parallel when multiple CPUs are available.
- Threads can overlap the execution of slow operations (e.g., I/O) with computational operations.
- DPI threads can share access to common resources, such as open files.
- Several programming models can be used for organizing DPI applications, including the *dispatcher worker*, *team*, and *pipeline* models [Tanenbaum, 1992].

2.3.6 Agent Languages

Elastic processing is language independent. That is, an elastic server can be configured to accept agents written in an arbitrary programming language. This section discusses the advantages and disadvantages of particular programming languages for writing delegated agents. Application programmers can write DP agents on different languages, according to their requirements and environment considerations. Programmers choose a given language for many different reasons: because they can reuse existing code, because its constructs simplify a given task, because of programming-culture bias, or to comply with administrative edicts. Some programs are easier to write in a scripting language, while others require industrial-strength languages and tools.

We designed elastic processing to leave the choice of agent programming language open[3]. We expected that newer and better languages would appear. And indeed, since we implemented the first prototype in 1991, several new agent scripting languages have appeared. Among the most popular are Java [Gosling and McGilton, 1995], Safe-TCL [Borenstein, 1994], and Telescript [White, 1994]. Each of these languages provides features that are useful for remote delegation.

However, many tasks cannot be effectively handled by interpreted "safe" languages. For example, operations management applications often require real-time tasks and direct memory access which are not supported by scripting languages. For instance, Telescript agents can not directly examine or modify the physical resources

[3]Therefore we avoided making a *"religious"* choice.

of the computers on which they execute. Similar restrictions are imposed by other *"safe"* scripting languages, like Safe-TCL and Java.

In theory, these languages can be used for almost any application. In practice, their limitations restrict their use to very specific application domains, such as executable e-mail messages (Safe-TCL) and animation applets in Web pages (Java). Section 2.7.4 presents a comparative description of these newer *remote scripting* technologies with remote delegation.

RDS allows programmers to reuse existing code and development tools to develop agents. Indeed, one can take any existing code and transform it to a delegated agent. An elastic process could be configured to accept delegated agents written in any of these languages. It is even possible to delegate an entire interpreter of a language L to an elastic process, and forthwith delegate agents written in L.

The first RDS prototype implementation supports agents written in a subset of ANSI C [ANSI, 1989]. This subset allows delegated agents to access a predefined set of functions, and eliminates their ability to invoke arbitrary external or internal functions. Even the default C library functions must be explicitly allowed by being listed in a special configuration file, which is loaded by the Controller.

We chose C as the base DP language because it is ubiquitous and general-purpose. Its low-level facilities, limited restrictions, and efficient implementations make it convenient for most programming tasks. Furthermore, we were able to reuse a lot of existing C source code to create agents. Each agent is checked by the Translator using the system linker. The output of the compiler is linked with an object file that contains empty definitions of the allowed functions. If the linker complains about undefined references, the DP is rejected and a proper explanation message is sent to the delegator.

Another option is to use a traditional programming language augmented with appropriate constructs for distributed programming. For instance, Concert/C [Auerbach *et al.*, 1994b] extends ANSI C to support remote process creation and communication via both RPCs and asynchronous messages. Interprocess communications interfaces are type checked at compile time and/or at runtime. All C data types, including complex data structures containing pointers and aliases, can be transmitted as parameters. Concert/C programs run on heterogeneous environments and transparently communicate over multiple RPC and messaging protocols. Concert/C hides the complexity of programming interacting distributed applications. It allows programmers to reuse their large existing base of code (and skills) in C.

DP Generation and Formats

Applications are not required to generate the code of their agents during execution. Most likely, a delegator process will use pre-defined and readily available agent programs. Such programs will encapsulate application-specific expertise. Still, nothing prohibits applications from on-the-fly construction of agents. For example, a Web server could dynamically compose a delegated agent based on the identity of the client. Such an agent could be written to match the computational resources

available to the client. For instance, if the client process executes in a wireless device then compression algorithms are built into the delegated agent. Alternatively, if the client host is a 3-D graphical workstation, special rendering effects are included.

Agents can be delegated in either source-code, intermediate-code, or object-code format.

- Source-code DPs require compilation and dynamic linking (C) or interpretation (TCL). Source-code DPs are portable and hence convenient to send to remote processes, as the sender does not need to be aware of the receiver's hardware architecture.

- Intermediate-code DPs are represented in a machine-neutral format that can be interpreted or translated to the local machine format. They require the existence, at the receiving process, of the corresponding interpreter or compiler back-end. For example, Java [Gosling and McGilton, 1995] compilers produce safe intermediate-code.

- Object-code DPs are in machine specific format. The delegator must be aware of the object code format accepted by the elastic process. As it is very difficult to evaluate the safety of a object code, it should require special authorization.

2.3.7 Execution Control and Reliability

RDS supports controlling the execution of remote DPIs. For example, a process may instantiate a remote agent that monitors certain device parameters, and reports periodically a computed statistical value. At certain times, an event of high priority may require that the delegator process temporarily suspend the DPI's reporting. RDS maintains the state of delegated agents in the elastic processes, i.e., an elastic process is not stateless. The code of the delegated agents is kept on stable storage, but the state of the thread instance is volatile by default. Delegated agents may choose to implement thier own recovery mechanisms to survive failures.

RDS must deal with different classes of failures, during both agent delegation and execution. For example, a delegated agent may not be accepted by an elastic process if the delegator process does not have the proper authorization. Also, a delegated agent may be rejected if it violates some safety constraint, it is too large to store, and so forth. During execution, there are many more potential error situations. For instance:

- a DPI may not be able to allocate sufficient memory for its computations,

- the delegator or the elastic process may crash during a message exchange, and

- network connectivity can be temporarily lost.

The problem of elastic process reliability is an instance of the generic problem of reliability for state-full processes. Thus, traditional recovery solutions, e.g., *transactions*

could be applied to RDS operations. The current implementation does not provide any support for recovery, takes trivial recovery actions whenever possible, and informs the parties when failures occur.

Implementation

The first elastic process prototype was implemented on the SunOS 4.1 O/S, using the lightweight process (lwp) library. This prototype was first demonstrated at the InterOp 1991 conference (see Section 4.6.5). The second prototype, the SMARTS Operations Server (SOS) [Dupuy, 1995], was implemented in Solaris and other operating systems.

2.4 Security

Network security problems can undermine the security of distributed applications. For example, (1) a protocol analyzer can compromise communications by analyzing protocol frames; (2) malicious attacks can target devices to bring an entire network down. Even simple statistical observations can be used to identify critical network resources and target them. *Security management* is concerned with monitoring and controlling access to the network resources [Stallings, 1993]. This Section describes some of the security issues of elastic processing, which are related to the above discussion.

2.4.1 RDS Security Requirements and Threats

Let us consider the specific RDS security requirements and the potential threats that they address. Traditional security requirements [Stallings, 1993] include: (1) information *secrecy or confidentiality*, e.g., authorized access to DP files, (2) *data integrity*, e.g., DPs can only be deleted by authorized parties, and, (3) *resource availability*, e.g., DPIs should not be denied to authorized parties. Potential security threats include *interruption* (e.g., an unauthorized party suspending a DPI), *interception* (e.g., accessing the contents of DPI messages), *modification* (e.g., DP tampering), and *masquerade* (e.g., assuming the identity of an authorized party to insert counterfeit DPs). Security requirements and threats must be addressed in the context of specific application and installations needs. Obviously an elastic process calendar application needs a high level of security at a military installation, and a lesser level at a high school.

RDS Introduces Potential Security Risks

RDS provides an *"access path"* to remote resources across security domains. Thus, RDS increases the potential risks of security exposures by providing opportunities for remote attacks. A few examples:

- A rogue agent could be delegated to an elastic process to destroy files. A trojan-horse agent could disrupt the execution of the elastic process, or even make it crash.

- A pernicious user could intercept and garner RDS frames en-passant. From these frames the attacker could infer information about the capabilities and location of a remote elastic process. They could, for instance, attempt to send a modified copy of a DP to the elastic process in disguise.

- A user could besiege an elastic process with requests, thereby denying access to other legitimate users.

Clearly, it is inappropriate for any process to import arbitrary code into a system without the owner's consent. Elastic processes should avoid receiving agents from non-reputed sources. DBM provides an explicit and configurable mechanism to receive and check imported code, based on authentication and constrained execution.

2.4.2 RDS Security Model

The RDS security model can resolve these problems. It includes (1) authentication of the original party making a request, (2) detection of tampering with a request, and (3) execution on a constrained environment.

Authentication

RDS clients need to be trusted to download and execute agents in remote entities. What these delegated agents are permitted to do may vary depending on the identity of the requesting party. For example, one subject may be permitted only to run certain predefined DPs, while another may be permitted to perform control operations. These privileges are defined by each elastic process. For instance, an elastic process that performs network management functions may only allow certain administrators to perform RDS actions.

The RDS protocol must be able to establish the identity of each requester. An RDS *party* identifies principal "subjects" which are the sources of RDS requests, and associates privileges with their identity. To determine who authored an RDS message requires an authentication framework which includes an unforgeable digital signature and a corresponding unforgeable digital integrity verification. At the same time, RDS must ensure the integrity of each request, i.e., that the contents of each message have not been tampered with. Data integrity assures that messages are received as sent, with no duplication, modification, or resequencing.

One way to achieve this is by *fingerprinting* each RDS message with a digital signature. This signature is a function of both the signer entity and the message data that implements each request. A message digest fingerprint can be used to guarantee that the data on an RDS message has not been modified. Data confidentiality assures that information is not disclosed to unauthorized parties. To support confidentiality,

the contents of the message must be encrypted, e.g., using the *Data Encryption Standard, DES.*

Message Digest - MD5

The MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit hash-code or *"message digest"* of the input [Rivest, 1992]. Rivest's conjecture is that it is computationally infeasible to (1) produce two messages that have the same message digest, (2^{64} operations) and (2) to generate a message that produces a given message digest (2^{128} operations). Thus, MD5 can be used to provide a digital signature of a message.

SOS authentication

The SOS implementation of RDS supports MD5 authentication. An SOS party uses the `ktadd` command to add security keys for the party to a key table. The key table entry for an SOS party identifies the authentication and privacy protocols used by the party. A party is identified by a string `<id>@<host>:<port>` where `<id>` is a party ID number and `<host>:<port>` identify the host and the port number of the SOS process. For instance, `ktadd 0@sutton:12345 md5Auth` identifies party 0 for the SOS at host sutton listening on port 12345, and requests MD5 authentication. The default is no authentication.

Access Control

Elastic processing can support a fine granularity of access control policies. For example, an elastic process can introduce access control lists that define the proper granularity of access to its local files and services. It can define lists of actions which are restricted, and determine the appropriate granularity of control for those actions. For example, a certain class of DPIs can be allowed to invoke a specific list of predefined services, e.g., read(), but will not be allowed to invoke others, e.g., write(). An elastic process could give to each delegated agent the capability for a file to be written. Thus, the elastic process will not risk having the agent write over other files in the same directory. A major problem with capability-based approaches is the complexity involved in keeping track of all the different capabilities.

The elastic process itself has limited access to local resources. These restrictions are enforced by the local O/S, e.g., by executing the elastic process under constrained privileges. A DP can not access resources that the elastic process itself could not have accessed. Thus, each elastic process establishes a security domain for DPIs. For instance, an elastic process in Unix can be executed with a userid of *nobody* which has no access to any local resources.

Example: Network Management Security

One of the main applications of elastic processing is network management. Network managers need substantial control over remote resources, e.g., they need to reboot devices and upgrade their software. Such actions are difficult to perform using a *"safe"* language based security mechanism. These language-based mechanisms typically restrict these types of actions, and do not provide built-in authentication. For network management purposes, an elastic process should prevent unknown parties from accessing any resources, while allowing the authorized managers full control over their devices. Therefore, as long as the elastic process can insure the authenticity of the party making the request, it should allow it to perform its management functions.

"False" security occurs when a supposedly strong security mechanism is implemented on a weak underlying foundation. For instance, a secure language interpreter (say Java) can give a false sense of security if the underlying O/S allows a virus to infect the interpreter itself. Security functions are effective only when they rely on strong O/S security foundations. It is wasteful to allocate significant resources for enforcing security, and then execute them on top of an O/S that has many security holes.

2.4.3 Safety

An elastic process can be configured to accept agents from different sources, creating potential security problems. For instance, a Web browser that imports agents from public non-reputable repositories incurs a risk of *"viral"* infection. A *safe* programming language is expected to guarantee that agents written in it can not harm the accepting process. For example, safe-TCL will execute script agents considered unsafe on a separate interpreter with restricted access, preventing them from accessing resources in an unauthorized fashion. An attempt by such an agent to access a file will result in prompting the workstation user for authorization.

Most general-purpose languages, like C, are inherently *unsafe*, and can expose the elastic process to unacceptable hazards. For example, a C agent can overwrite unintended memory locations within the shared address space, cause memory leaks, and so on. Unsafe agents can potentially cause the entire elastic process to crash. These risks may be unacceptable for many applications.

Safe languages must restrict the capabilities of their agents to prevent such problems. Hence, safety often comes at a significant cost in programming flexibility. Indeed, safe languages are unsuitable for many application tasks. Safe-TCL, for instance, is a small *"scripting"* language that lacks arrays and structures to make linked lists. Its numeric operations are interpreted and therefore too slow for many types of applications.

If agents can be written in an unsafe language, the applications that receive them will often require a stronger level of *security*. For instance, they will only accept agents from well known sources, after verifying that the agents are original and have not been altered. One way to achieve this is by *fingerprinting* each RDS message with

a digital signature.

2.5 Performance

This section evaluates the performance characteristics and implications of elastic processing for distributed applications. Our goal is to compare the performance of RDS applications to traditional C/S alternatives.

2.5.1 Background

Two basic metrics for evaluating the performance of computing systems are *response time*, the time between the start and the completion of an event, and *throughput*, the total amount of work done in a given time period. Our main concern is that of minimizing the overall response time or *transactional* time for the application end user who makes a request. Studies report that user productivity is inversely proportional to transactional time [Hennessy and Patterson, 1990]. The transactional time performance of a distributed application request depends on many factors:

- System parameters, such as CPU speed, memory size, and network bandwidth.
- Application dependent factors, such as the number and level of process interactions, the type and number of parameters for each exchange.
- Communication related factors, such as data marshalling and network quality of service requirements.

2.5.2 A simple example

Let us compare the response time performance of a simple distributed application, based on the TAA example of Section 2.2.4. This application consists of two sequential processes, a client executing at a laptop, and a server process executing at the home office. The client needs to retrieve archival e-mail messages which fit certain criteria. The message sequence $M = (m_1, m_2, ..., m_n)$ is stored on a large file. The filtering criteria on a message m_i is defined by an arbitrary Boolean function $FilterCriteria(m_i, m_p)$, where m_p is the previous e-mail message that fits the criteria. For instance, the FilerCriteria is looking for a chain of related e-mail messages.

A server process, S, provides file-level access to the e-mail file. For instance, let us first consider an interaction with the original `rmtd` server. The TAA client process will first open() the file, and then sequentially read() one by one all the file records. It will then execute $FilterCriteria(m_i, 0)$ until it finds a message m_j that matches the criteria. The m_j becomes m_p until the next message is found. The client will then examine each record by executing $FilterCriteria(m_i, m_p)$ on each message m_i, for $i = 1, n$. Those messages that match are saved on a sequence $(x_1, x_2, ..., x_k)$ which is needed by TAA.

The overall response time cost function for this rigid server interaction, T_R, can be approximated by

$$T_R = t_N(open) + n * [t_S(read) + t_N(read) + t_C(F)]$$

where $t_N(x)$ is the network delay of the remote interaction x, $t_S(read)$ is the time required for reading the record from the file, $t_C(F)$ is the time to execute the `FilterCriteria` operation in the client host, C. Assume, for simplicity, that n also corresponds to the number of file interactions required to read the file.

Let us now consider a similar interaction, but this time using an elastic server, `ES`. In this case, the client will delegate the code `FilterCriteria`, e.g., via `RDS_Delegate(ES, FilterCriteria, &DPid)`.
It will then instantiate a DPI using `RDS_Instantiate`. The DPI will send to the TAA client a copy of each message x_i that matches the filtering criteria.

The overall response time for this elastic server interaction T_E can be estimated by

$$T_E = t_N(del) + t_{ES}(del) + t_N(ins) + n * [t_{ES}(read) + t_{ES}(F)] + k * t_N(rec)$$

where $t_{ES}(x)$ indicates that the operation x is executed on the elastic server, and k is the number of messages that fit the given criteria. The operations del, ins, and rec represent the cost of the RDS operations of delegating the DP, instantiating the DPI and receiving each message.

Let us compare T_R and T_E. The cost of file reading is common to both so we can ignore it. For simplicity, we assume (1) that each C/S interaction takes the same time, τ, and (2) that there is no overlap between the actions. Thus, the rigid interaction includes $(n+1)*\tau$, while the RDS interaction has $(k+2)*\tau$. The remaining difference is in the location of the execution of `FilterCriteria`, and the extra cost of integrating it in the elastic server. In summary,

$$T_R = (n+1) * \tau + n * t_C(F) + t_S(read)$$

and

$$T_{ES} = (k+2) * \tau + n * t_{ES}(F) + t_{ES}(del) + t_{ES}(read).$$

A few observations are in order:

- The cost function t_N includes the time spent on activities such as marshalling data and the actual network transit time.
- The cost of computing the function `FilterCriteria` depends on fixed and variable factors. Fixed factors include CPU type and memory sizes. A variable factor is the processing load of a shared host at the time of the request. For instance, it is likely that a server host will be able to compute the filtering function much faster than the laptop.

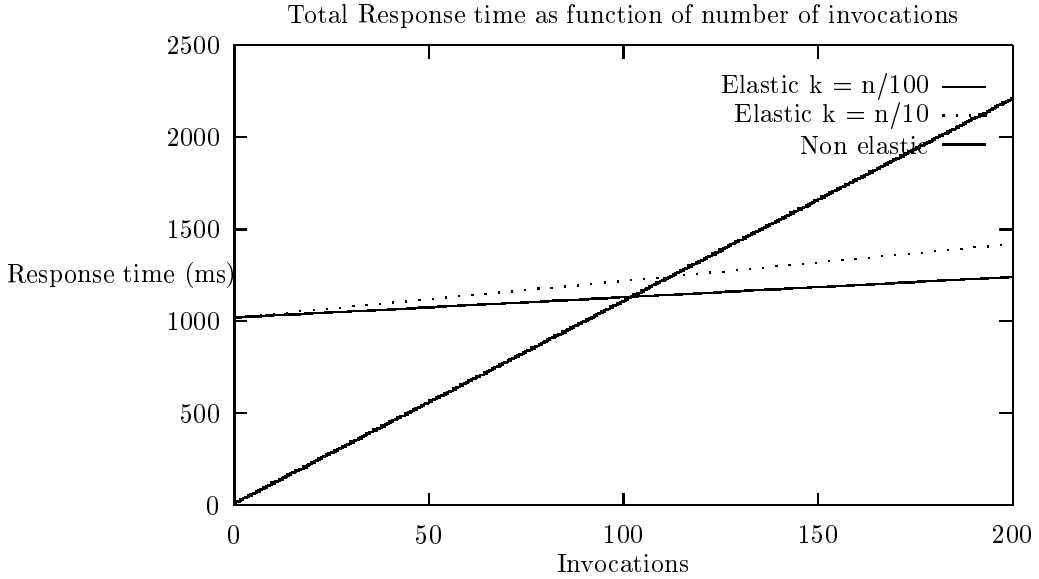

Figure 2.9: Aggregated Delay in LAN.

- The cost of delegating `FilterCriteria` into ES could be significant, but it is paid only once.

Let us assign reasonable numeric values to the above parameters to obtain the curves. The cost of integrating a DP into the elastic process depends on the host computer environment (CPU, compiler, and so forth). For a small DP, like `FilterCriteria` we can approximate it by 1 second. The cost of each network interaction varies depending on the network distance and the type of network. For a LAN, a typical interaction would take approximately 10 ms, for a MAN 100 ms, and for a WAN 1000 ms.

Figure 2.10 depicts the delay as a function of the number of executions of remote access functions for a WAN. Figure 2.9 depicts the same function for a LAN. The two lines for elastic processing correspond to $k = n/100$ and $k = n/10$. If the number of remote accesses is small, the overhead of delegation is relatively high, and a rigid process has smaller delay. As the number of remote interactions needed to retrieve data grows, the relative cost of delegation lowers. After a certain point, the aggregate delay is lower on a remote delegation interaction. Figure 2.11 depicts the delay on a LAN, considering a non-elastic interaction with a CPU which is four times slower than the server.

2.5.3 Performance Analysis

As computer processors get increasingly faster, distributed applications processes will spend more time waiting for their remote interactions. Partridge [Partridge, 1992] examines these efficiency tradeoffs by comparing several paradigms of

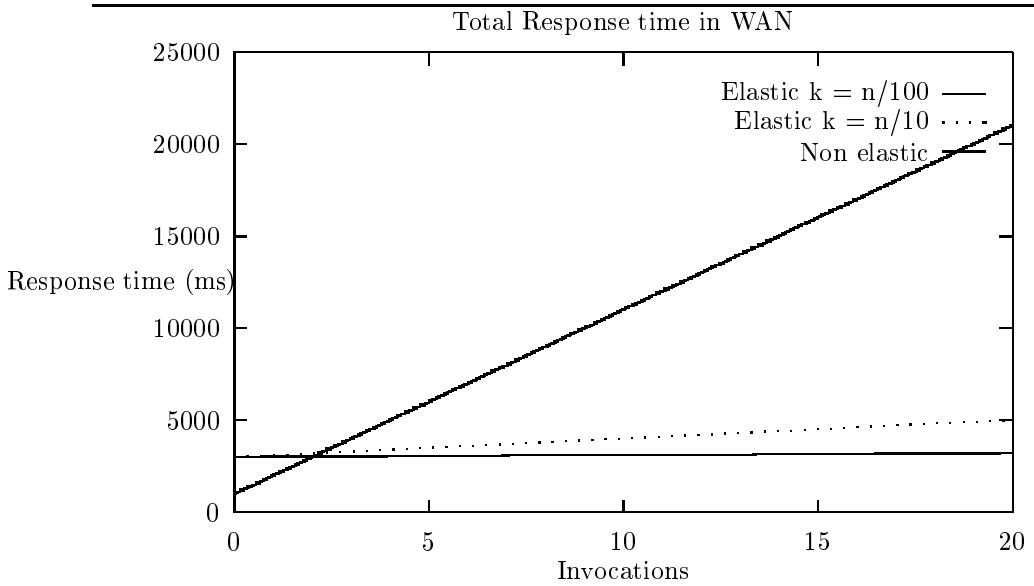

Figure 2.10: Aggregated Delay in WAN.

distributed computing interaction.

CPU cycles vs Network Latency

Over the last few years, CPU power has increased faster than any other computing technology. Researchers predict that the performance of microprocessors will continue to increase at an annual rate of 50% [Hennessy and Patterson, 1990]. Network bandwidth availability is also increasing, but at a slower rate and higher relative cost than CPUs speeds. Network latency, however, is limited by the speed of light, and quality of service constraints. Network delays are caused by geographical distance, physical layer technology and by contention for network resources. Depending on the network's topology and routing, the delay can be significantly different than what would be expected from purely geographic considerations. For example, the round-trip delay between two hosts in Austin, Texas was measured as 596 ms, while that between one of these hosts and a host in Japan was only 254 ms [Carl-Mitchell and Quarterman, 1994].

Tradeoffs

It is much easier and inexpensive to provide dedicated fast CPUs than to establish dedicated fast network connections to significantly reduce network latency between remote hosts. Even with the best networking technology, network latency remains constrained by the physical limitations of the medium[4]. Because network

[4] *"There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed - you can't bribe God."* David Clark, MIT.

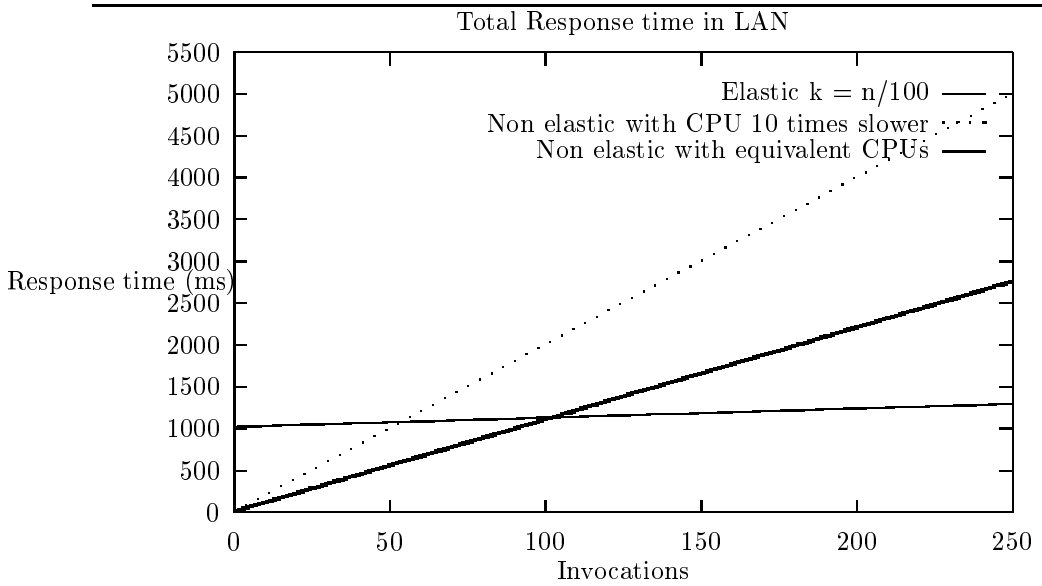

Figure 2.11: CPU effect on Aggregated Delay in LAN.

delay is not decreasing at a rate comparable with the speed increases of microprocessors, network latency will increasingly become the major performance bottleneck. In other words, as CPU processing speeds increase, more distributed tasks will spend more idle time waiting for their remote interactions to complete. That is, the relative delay experienced by applications, measured in CPU instruction cycles, will increase.

Distributed programs that follow the traditional C/S interaction paradigm will increasingly spend a larger portion of their time waiting for remote responses. Waiting for data to cross the network will become a larger part of distributed programs executions. Reducing the number of data exchanges over the network is, therefore, critical to improve the performance of distributed systems. Delegation to an elastic server provides a programming paradigm that can effectively reduce these exchanges.

Inverse Caching

Elasticity is an *"inverse-caching"* solution to the latency problem, i.e., to move the applications closer to where their resources are located. Data caching, in contrast, is used to keep data closer to where it is used. Data caching is most effective when there is a high degree of locality of use, and a high degree of "hits". Caching, however, cannot properly support highly volatile distributed data, which is quickly out of date, and makes the cache inconsistent. For example, caching the results of measurements from remote monitoring instruments is not efficient if their measurements change frequently. In such cases, getting code closer to the data is a more effective approach.

The spatial distribution of elastic processing provides a more efficient utilization of network bandwidth by eliminating many data exchanges. As network delays

will increasingly dominate, the relative cost of individual interactions increases. Obviously, delegation is only an improvement as long as the penalty of delegating code to another process is smaller than the gain obtained by shifting function closer to its data.

2.6 Applications of Elastic Processes

This section presents examples of application domains where elastic processes can be used. The examples illustrate specific problems of distributed applications, and describe how RDS can help application developers to address them. Section 2.6.1 focuses on dynamically extending the functionality of executing applications, and Section 2.6.2 describes the problems of adapting to varying computational resource availability. Section 2.6.3 briefly outlines some of the generic properties that characterize applications that can benefit from elasticity. Later chapters describe the application of elasticity to solve network management problems.

2.6.1 Extension of Applications

Some software applications provide limited capabilities for dynamic extensibility. For instance, spreadsheets and word-processors support scripting languages that allow application users to customize the application. To extend such an application, however, end users must learn to program in its specific language. A major drawback is that to reuse existing code typically requires manually translating it to the specific language. RDS allows delegated agents in different programming languages, provided the corresponding elastic process was properly configured.

The *World Wide Web* provides an appropriate framework to exploit the capabilities of delegated agents. For instance, delegated agents can be used to search unstructured information available in the Internet, to play interactive multiuser games, or to extend browsers with new multimedia capabilities.

Extending a Web Browser

Consider a distributed application that assists users in calculating tax scenarios. The application could dynamically extend a Web page of HTML tax forms with a small spreadsheat. We could build an elastic Web browser that accepts the spreadsheet as an extension. This extension could be written on whatever language the spreadsheet was written, if the elastic browser was properly configured to support it. The security of this application could be based on authenticating the source of the program as a reputable provider of such services, and executing the browser process in a constrained environment with only read access to certain files.

HotJava [Gosling, 1995] is an example of an extensible client browser that accepts code written in the Java language. Application developers can not easily reuse their existing code for such extensions. Interpreters for languages like Java or

TCL increase the memory size of the process. Also, the interpreted code typically executes much slower than compiled code. We defer a more detailed comparison with this technology to Section 2.7.4.

Extending a Web Server

Consider a laptop computer application to make reservations for a vacation, including hotel, car, tours, and so forth[5]. This application will support arbitrary personal constraints which are defined by rules to ensure the best possible vacation for a given budget. Assume that the relevant information is available at some remote Web server. A non-elastic Web server allows its clients to retrieve information based on predefined queries, e.g., using forms. However, the Web server could not have predefined all the possible types of queries for each vacation. The application will need to retrieve large amounts of data from the server, and filter it locally at the client host. It will then execute the booking transaction at the server.

This scheme is probably inefficient, insecure, and expensive. It is inefficient since it wastes CPU cycles of the client and server. By the time the information has been retrieved and filtered, it may no longer be relevant, e.g., the hotel room may have been taken. It is insecure because the server must expose a lot of data to the client. The server owner may prefer not to expose some of this data for competitive business reasons. If this transaction is being performed over a wireless or long-distance phone line, the cost of the data exchange is expensive.

An alternative solution is to dispatch delegated agents to an elastic Web server. The delegated agent will contain all the specific rules and constraints for a particular transaction. Such an agent could be delegated to servers from competing organizations, prompting them to compete on real time. Using delegated agents, applications can overcome many resource constraints. For instance, bandwidth limitations are avoided by reducing the transfer of unnecessary data.

Extending an E-mail Based Service – DFlash

The *dFLASH* server [Califano and Rigoutsos, 1993] is a homologous sequence retrieval program for protein sequences. The server supports remote researchers via e-mail requests. These requests include directives to: (1) retrieve data contained in a very large database, e.g., about authors and dates, (2) restrict the number of reported sequences and alignments, and (3) restrict the number of reported sequences to only those whose score exceeds a given threshold value.

The services provided by the dFLASH server are fixed. Researchers need to send several e-mail messages to the server, and wait for their e-mail answers to retrieve all the sequences that they need. Then, they need to locally select and filter the retrieved data. Using e-mail as the data exchange protocol has the advantage of ubiquitness, and the disadvantage of very high delays. Using RDS over e-mail, a

[5]This example was described in Section 1.2.1.

dFLASH server could be extended to support complex queries. For instance, a DP could be sent to perform queries that combine selective criteria about authors and their sequences.

Software Upgrading

Designers of long lived distributed applications cannot anticipate all the functional upgrades and customizations that will be required after their software has been installed at each site. Upgrading software typically requires shutting down processes and replacing them with newly compiled versions. In many cases, e.g., phone switches, the expenses of bringing the system down are substantial. Elastic processing enable software upgrades during execution without requiring rebooting.

Software Customization

Many devices can be dynamically customized using RDS. For instance, consider the programming of an interactive game for a top-set TV device. RDS can be used to to customize the game's processes to adapt to different levels of players. The application could dynamically replace, add, or remove specific algorithms. More generally, any software vendor could use delegated agents to customize their software. For instance, a network service provider could dynamically delegate agents to interactively help new users to learn how to use an application. Such an agent could be developed based on the experiences of other users, after the original application has been deployed.

Software Monitoring and Debugging

Another use of delegation is to augment a distributed application with monitoring capabilities. This can be very useful for debugging distributed applications [Goldszmidt *et al.*, 1990]. A DPI thread can be instantiated to monitor and trace the occurrence of certain events on distributed processes. For instance, DPIs could be used to discover patterns of data access, collect and correlate event traces, and so forth.

Application Interoperability

Many distributed applications must support an increasing number of protocols to provide interoperability with different applications. For example:

- A bank server may need to support different versions of a cryptographic protocol to comply with legal regulations that restrict software exports.
- E-Mail browsers need to support several formats to handle messages of different types.
- RPC servers may need to support (directly or through stubs) different data representation protocols, e.g., `NDR`, `XDR`, and `BER`.

Providing fixed support for all possible combinations of protocols and data representation formats results in large additions of code, whether it will be used or not. For example, consider again the laptop client process that retrieves archival e-mail messages (Section 2.2.4). From time to time the client application may need to retrieve messages which were encrypted with some arcane algorithm. The client could dynamically retrieve the decryption code only when needed, thus saving premium laptop disk space.

Application Gateways

Elastic processing is an appropriate framework for building *application gateways* to support application interoperability in heterogeneous distributed systems. An application gateway performs the role of translating between different application-level protocols. For instance, an e-mail gateway can translate between different e-mail protocols for different mail processes. Elastic processes are able to dynamically incorporate software gateways to interoperate using different protocols. Dynamic software delegation reduces the size of the code required for conversions in heterogeneous systems. Transferring code when and where it is needed enables application developers to build more modular and smaller applications.

2.6.2 Adaptation to Resource Availability

Emerging networked environments will have a large number of small devices. For example, a *Personal Digital Assistant*, PDA, is a hand-held device which offers wireless communication with limited storage capabilities and small displays. Typical applications provide organizer functions (e.g., calendar), with communication abilities (e.g., e-mail). Another type of resource-constrained environment is a "topset" device that controls the user interface to an interactive television. These devices have very limited memory and their network connections have limited bandwidth.

Many distributed applications have been designed and implemented with an implicit assumption that both clients and servers will run on powerful hosts. As PDAs and other small devices proliferate, users will request to execute their favorite applications on these devices. A typical Web browser client, for instance, can take up to 1MB of storage. However, the amount of code actually used depends on the particular navigation path followed by the user. A PDA or small laptop computer may not be able to keep all of the process code locally, due to memory and storage limitations.

Dynamic extensibility could allow such a process to run with a minimal functional core, and retrieve, when needed, more code from a remote server. This is akin to extending virtual memory over the network. Elastic processing allows application designers to implement applications that can be dynamically adapted to limited computational and/or bandwidth resources. Note that such resource restrictions may be intermittent. For example, the amount of available storage on a desktop host may be limited when a disk becomes full, or the available bandwidth may be limited

for a period of time. Elastic processing allows applications to shift functionality in accordance with the availability of computing resources.

Example: Real-Time manufacturing

Consider a factory floor process control environment. Within a manufacturing line, a device may need to react to a fault in real time. If the logic to handle the fault is centralized in a client, it will require a few interactions for diagnostics. A corrective action may miss deadlines due to unpredictable network delays. Elasticity enables a client to delegate to the device the instantiation of time critical functionality, reducing the need for interaction.

Example: Bandwidth Constraints

Consider a centralized client that interacts with many networked devices on several ethernets, e.g., a fire-alarm process that monitors temperature readings by polling. As the number of devices grows, bandwidth becomes a bottleneck, particularly during high network load. Delegating monitoring to a hierarchy of processes during high load times reduces the number of ethernet packets. RDS will allow the client to reduce its bandwidth requirements and therefore be able to control more devices. RDS can assist in reducing the resource requirements of an application as conditions change over time; For example, when the network becomes congested, and when configuration changes as applications and hosts are introduced and removed.

2.6.3 Generic Properties

The following properties characterize classes of applications that can benefit from elasticity:

- They execute on a dynamically evolving heterogeneous distributed environment to which they must adapt, for example, as new hosts and devices with different hardware architectures are incorporated to the distributed system.
- They execute on a dynamically changing administrative environment. For example, security restrictions on accessing devices may be imposed during certain periods, e.g., long distance calls are not allowed from departmental phones after office hours.
- They execute over hosts with insufficient computing resources, e.g., CPU cycles and/or main memory. For example, they execute over small devices with limited available RAM or diskless workstations.
- They execute over low bandwidth networks. For example, over phone lines or wireless connections.

- They are sensitive to latency delays, e.g., they need to respond to remote events in real time.
- They must sometimes deal with intermittent resource restrictions, for example, when a swap disk area becomes full, or when bandwidth is limited on a wireless connection.

2.7 Related Work

Distributed applications are designed and implemented following diverse processing and communication models. Examples of these include *Remote Procedure Call*, *Remote Execution*, *Remote Evaluation*, and *Remote Scripting*. This section reviews these models of distributed processing, and compares their relevant features with remote delegation to elastic processes.

2.7.1 Remote Procedure Call – RPC

RPC [Birrell and Nelson, 1984] is a widely used mechanism for distributed IPC. A critique of the RPC model is presented in [Tanembaum and van Renesse, 1988] and a survey of RPC systems in [Soares, 1992]. In the RPC model a server exports a number of fixed procedures that can be invoked *synchronously* by remote clients. Upon a remote call, the caller is suspended, the remote procedure is executed by the server, and its results are returned to the caller, which then resumes its execution. The following are some of the differences between an elastic server and a traditional RPC server.

RPC procedures are statically defined

RPC procedures must be statically defined at the time the server is compiled. Server designers must write all the procedures required to support the entire range of services for which a particular server may be invoked. Server designers must have foreknowledge of (or try to predict) all the possible scenarios in which the server can be involved. For many distributed applications, however, such early prediction is typically not possible. In contrast, the definition of a delegated agent procedure for an elastic server is independent of the server's design time. It can even be defined while the server executes.

RPC is synchronous and client initiated

A second difference is the synchronous, client initiated invocation semantics of RPC. In many applications, the invocation and execution of a server's procedure should be tied to events that are not under the client's control. For instance, a fault handler procedure on a server may need to be invoked as soon as the fault is detected in the server's host. Using an RPC, the client would block until the occurrence of such an

event, and then until the completion of the remote procedure. In contrast, a delegated agent procedure can also be invoked without client intervention. For instance, it can be triggered by an independent event in the elastic server host, asynchronously with the client's execution.

RPC typically requires several data exchanges

A third difference is in the amount of data transferred between the processes. Typically, RPC clients and servers exchange data parameters in several RPC interactions. For example, consider again the application that retrieves records from an archival tape. The client will invoke a `read()` RPC of the `rmtd` server several times until it finds the appropriate record. Each invocation of `read()` retrieves a block of data from the tape file. This interaction pattern results in an increased overall delay due to network latency.

In contrast, a typical elastic processing interaction involves significantly fewer network data exchanges. An elastic server version of `rmtd` would get a delegated agent to perform the search in one network message. It will then return to the client only those records that match the given criteria. Thus, RDS can reduce the number of data exchanges, as the computation is performed in the server.

Asynchronous RPC

One way to improve the performance of RPC calls is to make them non-blocking. This can result in efficiency gains if the client can perform some useful work *while* the RPC executes concurrently. Many research efforts have tried, with limited success, to address the intrinsic performance limitations of RPC. *Asynchronous* RPC (A-RPC) supports non-blocking RPC calls. *Promises* [Liskov and Shrira, 1988] is an example of an A-RPC design. The Concert/C [Auerbach *et al.*, 1994b] language supports both RPC and asynchronous message sends. In an A-RPC, the client has the choice of either wait for the call to return or to continue execution concurrently. If it continues execution, the client can later check the return status of the remote call. If it has not yet returned, the client has again the choice of blocking until the RPC returns, or continuing. Note that A-RPC does not maintain the simpler semantics of procedure invocation, which is the main reason for the popularity of RPC.

2.7.2 Remote Execution and Creation of Processes

Remote Execution

A remote execution facility allows users to execute processes on remote hosts. However, the executable modules must already be present in that host. The V-system [Cheriton, 1988], for instance, allows programs that do not require low-level hardware access to be executed remotely. For example, a program `p` can be remotely executed from the command interpreter on a randomly chosen host by typing: `<p><arguments>@*`. Another example is the NEST system [Agrawal and Ezzat,

1987] which provides a location independent remote execution facility. The main use of these and other similar facilities is for load balancing, i.e., utilizing remote CPU cycles, in homogeneous distributed operating systems.

Remote Process Creation

Other forms of remote execution are used to develop distributed applications across heterogeneous environments. For instance, Concert [Yemini *et al.*, 1989] introduces a language-level remote process creation construct, `create`. The `create` primitive instantiates processes at remote (or local) hosts, allowing them to exchange bindings and communicate. To create a remote process, the executable module must already be at the site where the process will execute. A Concert program is more portable, since its runtime implements a common software middleware that handles the low-level system calls to the different operating systems. Thus a Concert/C program does not need to invoke explicit process management services to create processes on hosts with different O/S. However, Concert processes are not first class. That is, they can not be exchanged as parameters in RPC calls.

2.7.3 Remote Evaluation and Process Migration

Remote Evaluation allows a program code expression to be transferred between hosts for execution. Thus, it is a restricted form of elasticity that combines delegation and invocation into one single action. Typically an interpreter at the remote host evaluates a program expression and returns the results to the client. REV [Stamos and Gifford, 1990b], SunDew [Gosling, 1986], and NCL [Falcone, 1987] all implement variations of remote evaluation mechanisms. REV supports remote evaluation of program expressions written in the CLU language [Stamos and Gifford, 1990a]. NCL [Falcone, 1987] was used to implement a networked file system, using remote evaluation of LISP expressions. Partridge [Partridge, 1992] argues that *late binding* RPC, a form of remote evaluation, gives optimal performance in the number of network transits required to complete a computation.

In remote evaluation interactions, the execution of the evaluated program is synchronous. A REV program is executed upon its receipt by the remote host interpreter. Furthermore, a REV client has no control over the remote program execution. For example, the client cannot cancel the evaluation of a REV expression. Finally, the design and implementation of remote evaluation is programming language and machine specific. That is, programs can be exchanged only between two computers of the same architecture that support the same language environment.

Delegation to an elastic server is a process-to-process communication, while REV is a computer-to-computer communication. A delegated agent procedure becomes an integral part of the receiving elastic process, so that it has access to the process internal scope and to its external environment. A remote evaluation procedure is self contained.

Remote evaluation goes a step further than RPC in permitting the dynamic transfer of programs to remote execution. However, both RPC and REV models involve synchronous procedure-call interactions whereby the agent executes the management program upon invocation. Then, the caller is blocked until the completion of the invocation, and/or it cannot exercise any control over it.

Process Migration

Process migration[6] applies load sharing and balancing algorithms to distribute workload among processors. The goals and implementations of elasticity and migration are different. Process migration focuses on obtaining efficient hardware utilization, via low level mechanisms that move binary images between computers. Thus, it is applied in homogeneous environments, in terms of same hardware architecture and operating systems [7]. In contrast, elasticity supports a high-level, application-oriented communication model.

2.7.4 Remote Scripting with Safe Agents

Scripting agents have recently attracted great interest as a candidate technology to provide extensible and *"intelligent"* networked services. The word agent is being broadly abused to refer to many different entities [Riecken, 1994a]. Some authors define an "agent" as any software program that acts on behalf of some other entity or user. For instance, *"Intelligent"* agents [Riecken, 1994b] is an active area of research whose main focus is in artificial intelligence issues, such as knowledge representation and planning.

Mobile agents are programs, typically written in a scripting language, which are dispatched to a remote computer for execution [Harrison *et al.*, 1995]. Remote *"scripting"* agents enhance the remote evaluation model by providing safe languages and security features. Several scripting languages have been proposed and used to write *mobile* agents, e.g., Java [Gosling and McGilton, 1995] and Safe-TCL [Borenstein, 1994], *"Itinerant"* agents are scripting agents which roam among a set of networked servers, seeking assistance and collecting information [Chess *et al.*, 1995]. Telescript [White, 1994] is an example of an itinerant agent technology. The following paragraphs outline their main characteristics.

Java

Java is an interpreted, multithreaded and "type-safe" dialect of C++. Java removes from C++ *"unsafe"* features, e.g., its pointer model eliminates the possibil-

[6] A survey of process migration mechanisms is given in [Smith, 1988].

[7] A design for supporting heterogeneous process migration is proposed in [Theimer and Hayes, 1991]. It consists of building a machine-independent migration program that specifies the current code and state of a process to be migrated. The migrating program is then recompiled, its state is reconstructed and execution is then continued on the target machine.

ity of overwriting memory and corrupting data. Java also removes what its author considers *"unnecessary"* features of C++, such as operator overloading, multiple inheritance, and implicit declarations of methods. The Java interpreter includes a *safety* component that verifies that the language rules have been respected. But the Java interpreter provides no real security against "trojan horses" and other improper behaviors. Java *applets* are used to add dynamic content into Web pages. These applets can be rendered by a Web browser that supports a Java interpreter, e.g., HotJava [Gosling, 1995]. The HotJava browser supports different levels of network access control.

Safe-TCL

Safe-TCL is a TCL [Ousterhout, 1990] dialect that removes from TCL all the features that can be "harmful" to the recipient. It replaces them with a number of commands that give untrusted programs a limited ability to interact with the user and the environment. Thus it permits mail applications to deliver active messages containing programs that can interact with their recipients. Safe commands are used to store and retrieve persistent data, send mail and print data, after the user's consent is obtained. The implementation of Safe-TCL is based on a *"twin interpreter"* model. The Safe-Tcl process contains two interpreters, a trusted interpreter and an untrusted interpreter, which have a relationship analogous to user space and kernel space in Unix. An application of Safe-TCL is the execution of e-mail messages.

Telescript

Telescript *"messaging"* agents scripts are downloaded to remote interpreters called *"engines"*, which are located at *"places"*, stationary objects which represent locations, e.g., a PDA. Agents can transport themselves between places and interact with other agents in remote places. The Telescript language interpreter precludes programs from accessing facilities from directly examining or modifying the physical resources of their places. A *permit* is an object that prevents its holder's to use certain instructions or limits the amount of a resource it can use. For instance, an agent may be denied the use of the travel instruction `go`. Agent permits are created explicitly, and must be renegotiated when the agent travels.

2.7.5 Comparison with Remote Scripting Agents

Remote scripting agents are programs which are transported from a client host to a server host for execution. Remote delegation offers a few significant advantages over remote scripting. The following paragraphs contrast some of their features with those of elastic processing. Table 2.1 presents a taxonomy of agent technologies.

The above technologies require that their agents be written in their particular language. This introduces the problems described in Section 2.3.6. Elastic processing

Name	RDS -MbD	SOS	Telescript	Safe-TCL	Java
Implementation	1991	1994	1994	1994	1995
Languages	independent	independent	Telescript	TCL	Java
Code Compiled	yes	yes	no	no	partial
Interpreted	no	optional	yes	yes	yes
Authentication	trivial	MD5	permits	no	no
Safety	optional	optional	yes	yes	yes
Low-level access	yes	yes	no	no	no
Scheduling	explicit	explicit	no	no	no
Agent Dispatch	yes	yes	yes	yes	no
Agent Retrieval	yes	yes	no	no	yes
Main application	Network management	internet services	electronic commerce	active e-mail	HotJava animations

Table 2.1: Comparison of Agent Technologies

leaves the choice of agent programming language open. New programming languages for scripting agents do not support legacy code and programming skills.

This is not an impediment for new applications, particularly those tailored to a specific application domain. For example, Java has been used for developing Web page animations, and Telescript has been used to develop multimedia mail applications for personal communicators. For many application domains, however, adopting a new language requires a major effort. For instance, a new language requires rewriting existing applications, retraining programmers, acquiring or adopting new tools and development environments, and so forth. Furthermore, some languages are simply inappropriate for specific tasks. For instance, Telescript is *not*, as claimed, an *"ideal vehicle for network management"* [White, 1994].

In contrast, delegated agents are programs written in arbitrary languages. RDS allows programmers to reuse existing code and development tools to develop agents. Indeed, one can take any existing code and transform it to a delegated agent.

Agents can be written in both compiled and interpreted languages. In particular, one can even delegate an interpreter for an arbitrary language, and then delegate scripts to it. This accomplishes greater generality, code reuse, efficiency, and range of applicability than moving to a new programming language.

Many tasks cannot be effectively handled by an interpreted language. For example, network management tasks often involve real-time interactions. Telescript, for instance, has no support for transactions, synchronization, or any other real-time interactions. Telescript agents can't directly examine or modify the memory, file system, or any other physical resources of the computers on which they execute. Similar restrictions apply to other safe interpreted languages.

Language-enforced safety restrictions limit the types of applications that can be supported by each language. In contrast, delegation agents can be efficiently linked

and executed as integral components of an elastic process environment. Elastic processes can be configured to allow only tailored access to the underlying facilities that they need. Other limitations stem from the implementation of the virtual machine where the scripts execute. For example, the current implementation of Java contains a garbage collector thread that is invoked when memory is depleted. At that time this thread preempts all other threads from executing.

Efficient interaction between scripting agents and other software at local or remote components may be difficult and inefficient. For example, Telescript agents negotiate with their interpreters for access to local resources. Each agent must explicitly handle any negotiation failures, e.g., resource depletion, etc. In contrast, remote delegation permits effective control of agent execution. Delegated agents can efficiently and transparently interact with each other, and access the resources of the elastic process. An elastic process may choose to implement any resource allocation policy, and can simply choose to give all delegated agents equal access.

A language-based approach requires more complex, explicit mechanisms to accomplish security. For instance, Telescript agents need to explicitly get involved in negotiating security permits when moving. The security model of Java and Safe-TCL are based on a safe language. All these solutions require a *"one-size-fits-all"* security policy, which often complicates the development of applications, even when security is not a concern. In contrast, RDS enables the configuration of a customized security model for each elastic process. An elastic process may choose to use an authentication mechanism for parties like MD5, or no authentication. In the same way, an elastic process could request a privacy protocol like DES to encrypt the contents of its messages.

Telescript assumes a single common currency scheme based on teleclicks, and always records all data in persistent store. Each agent is granted a maximum lifetime span measured in seconds, a maximum size measured in bytes, and a fixed *allowance* measured in *teleclicks* for expenditure of resources. If an engine exceeds any of its fixed allocated limits, it is destroyed without any grace period. These mechanisms are tailored to a specific application domain, e.g., *"the electronic marketplace"*, but are wasteful and inappropriate for other application domains. For example, for some management computations there may not be any way to predefine a fixed allocation of teleclicks. Other applications may need to execute on resource constrained environments, where there is no possibility of storing all data on disks.

Telescript, for instance, assumes the existence of a reliable transport between interpreters at remote places. This is not an acceptable precondition for many distributed applications. For instance, network management applications must function over unreliable transports. RDS does not require a reliable transport mechanism.

The execution of script agents cannot be remotely controlled. For example, suppose one site wishes to delegate agents whose execution may be controlled by other sites. Java scripts could not easily support this model. They would need to explicitly program and execute a master agent that simulates O/S functions. But because of the language safety constraints, such functions are not accessible to Java agents. At

best, these agents could voluntarily respond to remote commands. Elastic processing provides explicit remote execution control over delegated agents.

Finally, many of these scripting technologies assume a "closed-world" environment. For instance, Telescript agents can only *"extend the functionality of communication services to which they have access, provided those services are implemented using Telescript technology"* [White, 1994]. Similarly, Safe-TCL scripts execute in a constrained interpreter. In contrast, remote delegation agents can interoperate with other environments. Delegation does not force one or another inter-agent communications model, i.e., agents may use RPC, or any other model of process communications. This generality is important since it comes with built-in support of rich communication structures.

2.8 Conclusions

Remote dynamic process extensibility is a very useful feature for many types of distributed applications. Elastic processing with delegated agents presents a language independent paradigm that supports temporal and spatial distribution. This paradigm is defined by the RDS service and the architecture of the DBM runtime. The security model of elastic processes is based on party authentication and controlled execution. Elasticity is an effective paradigm to overcome the network delays which are a major performance bottleneck for many types of distributed applications. Elasticity assists application developers to address several generic problems of distributed computing environments.

3

Management by Delegation

Management by Delegation, M_bD, is the application of elastic processing with delegated agents to network and systems management. The thesis underlying M_bD is that efficient, reliable, and scalable network management systems can be developed using elastic processes as network management servers. Instead of bringing data from the network devices to the applications on a central platform, M_bD applications delegate parts of the management applications themselves to the networked devices. We begin this chapter by providing a background overview of current network management systems and a critical analysis of their problems. We then proceed to present the main results of M_bD.

Chapter Organization

Section 3.0 outlines the main functions of standard network management systems and presents a critical analysis of their problems.

Section 3.1 outlines the approach of M_bD to network management.

Section 3.2 describes examples of management applications that require decentralized solutions.

Section 3.3 outlines the design of M_bD and describes its components.

Section 3.4 describes the integration of M_bD within a standard framework, SNMP.

Section 3.5 examines some of the problems of network management systems and how M_bD addresses them.

Section 3.6 presents some conclusions.

3.0 Network Management

3.0.1 Background

The main goal of network management systems is to ensure the quality of the services that networked elements provide. To achieve this, network managers must monitor, control, and secure the computing assets connected to the network. For example, network management aims include the detection and handling of device failures, performance inefficiencies, and security compromises. To achieve these goals, network management systems implement specific management functions. The OSI standards have classified these functions into five major areas [ISO, 1989]: fault, accounting, configuration, performance, and security. Management applications use network management services to implement these functions. For instance, they collect management data using device instrumentation and management protocols, and present it to the operators via graphical user interfaces. To accomplish these goals:

- Management applications retrieve real-time data from network elements. For example, they collect the number of packets handled by a given interface of a router.
- Management applications interpret and analyze the data collected. For instance, they recognize security events, such as repeated illegal attempts to login on a workstation.
- Management applications present information to authorized network operators. For example, an application displays a network topology map with graphical representations of current network traffic.
- Management applications proactively react, in real time, to management problems. For instance, an application will disable a link that is experiencing faults.

These activities are implemented using protocols and data structures that follow standards guidelines. This section outlines the main characteristics that these frameworks share. Our aim is to give the minimal background that is necessary to understand the main technical problems of these frameworks. For a more comprehensive description of network and system management issues see [Sloman, 1994; Stallings, 1993].

Network Management System Architecture

Figure 3.1 shows a diagram of the organization of a typical network management system. The management platform consists of one or several workstations that interface with the network operators. Typically, an "umbrella" application provides a graphical user interface that integrates the user interfaces of many independent management applications. These management applications perform specific management

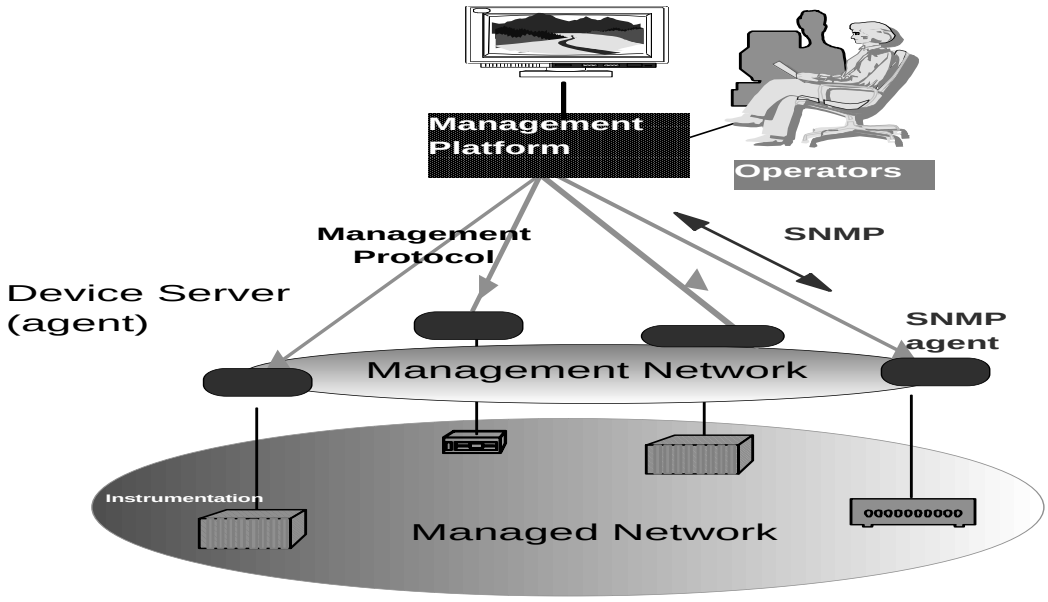

Figure 3.1: A Generic Network Management System Architecture

functions such as those described in Appendix A.2. The application processes that execute such functions assume a *"manager"* role.

Management "agents"[1] or *"device servers"* execute in managed network elements such as gateways and hosts. These servers are software processes embedded within each manageable network entity that monitor, control and collect data from their devices. These servers collect device data in *Management Information Bases* (MIBs) and support a management protocol. For instance, an SNMP-agent is a device server that implements an SNMP MIB. SNMP is an example of a management protocol used by platform applications and devices to exchange management data. Manager processes and device servers form part of a virtual *"management network"* which performs management functions over the *"real"* or *"managed" network*, as shown in Figure 3.1. The managed network consists of all the networked devices which have some type of instrumentation that permits their management.

[1]The network management community uses the term "agent" to refer to a server process at the managed device which supports a network management protocol and performs certain tasks. We will either use the term *device server* or qualify the name, e.g., SNMP-agent, to avoid confusing them with delegated agents.

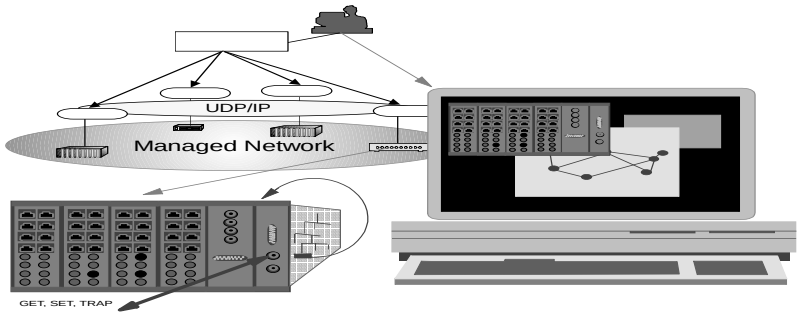

Figure 3.2: An SNMP Network Management System

Simple Network Management Protocol

SNMP is a standard network management protocol, whose name became a synonymous for the entire IETF network management framework. An SNMP network management system consists of a manager platform, managed nodes (each containing an SNMP-agent), a management protocol (SNMP), and MIBs. Figure 3.2 shows a manager platform with a GUI and a hub device that implements an SNMP-agent. The management network consists of SNMP-agents communicating via UDP. An application executing at the workstation displays a graphical representation of the hub. Thus, when the device indicates a port failure, the application can change the color of the corresponding port to red to indicate a problem.

In SNMP, management logic is performed in a central station on data collected from physically separated devices. Management applications are SNMP clients that contact one or more SNMP-agents. Each SNMP-agent is an MIB server that exports managed variables such as error counters and routing tables. SNMP is a polling-oriented protocol that uses a *fetch-store* paradigm. SNMP defines the syntax of the following management request messages: `Get` retrieves the current value of an MIB variable.
`GetNext` retrieves the value of MIB variables in lexicographic order.
`Set` modifies the value of an MIB variable.
`Trap` is a notification issued by an SNMP-agent as an unconfirmed asynchronous message.

MIBs are organized as static directory trees with managed data stored at tree leaves. The tree structure provides unique identification of managed data, which the protocol uses to read and write MIB data. To retrieve the value of a specific MIB variable, a `Get` command needs to identify the path on the tree to that variable. For example, `ipInReceives` is an MIB variable that counts the total number of IP input datagrams received.

3.0.2 Critical Analysis of Network Management Systems

Current network management systems follow a *centralized, labor-intensive* interaction paradigm. In this paradigm, a network management station polls operational data from network elements and displays it to the operations staff. Thus, the application programs resides in platform hosts while managed objects data resides within the networked devices. For example, an SNMP-agent maintains a counter of the number of packets that could not be transmitted by a router because of errors. A network management application at the NOC will poll the router to retrieve these counters and then compute and display the error rates. Operators must analyze this data to identify the causes of the errors, and diagnose, isolate, and correct the problem.

There are many types of network management applications that cannot be effectively addressed by these platform-centric frameworks, and require dynamic decentralization. We illustrate these through the following examples.

Example: Diagnosis of ATM Switches

A Network Operating Center (NOC) needs to diagnose and handle problems in its remote ATM switches. An NOC operator invokes an application program P that executes at a central host computer to perform the required management task as a client of the remote switch. The application uses SNMP to poll operations data from the switch and transmit to it commands. The switch contains a server process (an SNMP-agent) that exports predefined diagnosis and control services to external management applications. These services are *instrumentation* routines implemented by the device vendor. P uses polling to continuously retrieve diagnostics data from the remote switch, and then it analyzes the data locally. For instance, P may retrieve frame error counters to compute the standard deviation of the error rate. When the error rate statistics exceed a predefined threshold, P remotely invokes a corrective control action at the switch. For instance, it may request the switch to reset some interface cards.

Control Loop

The standard management paradigm establishes a *"control loop"* that involves the collection of monitoring data at the device, human interpretation and analysis of the computation at the NOC host, and the invocation of corrective actions at

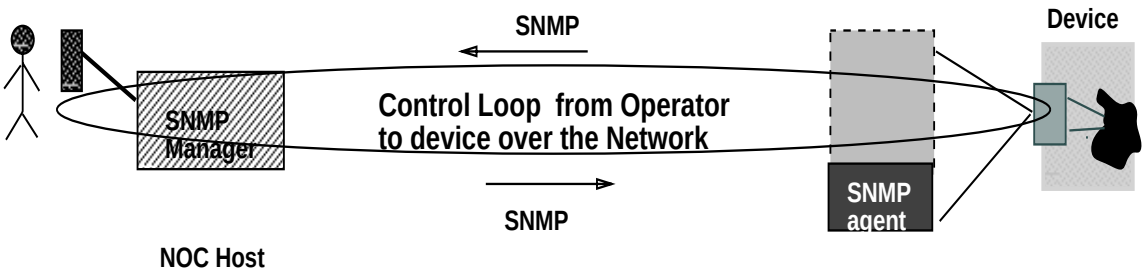

Figure 3.3: Control Loop over the Network in SNMP

the device. Figure 3.3 shows how this paradigm stretches the control loop from the managed devices across the network to the central hosts at the NOC. These control loops may fail when the network experiences failure problems, just when they need to operate best. Centralization thus seriously limits the reliability of a network management system.

Example: Automating the Management of Routers

Consider an organization that wishes to automate the management of its routers. That is, the organization wants to deploy programs that (1) monitor the operations of the routers, (2) analyze their behaviors and (3) invoke appropriate control functions. For example, suppose one wishes to deploy programs that monitor routing tables to detect routing problems and invoke appropriate handlers. When the network is large and fast, remote polling of large routing tables may consume significant bandwidth resources. The NOC hosts may be unable to detect and handle remote problems sufficiently fast. Centralization thus seriously limits the scalability of a network management system.

To facilitate the automation of management functions it is desirable to localize the monitoring and control at the devices. This could be accomplished by delegating management programs from the central hosts to the routers. The programs would then be linked with the local instrumentation to monitor, analyze, and control the routers. Upon detection of a problem these agents can cause the delegation of additional agents to the router and to attached systems and thus control their execution. These agents may be used to execute tests that isolate the problem and then reconfigure the router and other systems to recover from it.

Standard Approaches to Network Management

The above application examples illustrate a typical network management interaction following the prevailing standards paradigms. Network elements are equipped with instrumentation and control services that execute in the scope of the device itself. These services are defined by the device vendor and/or by standards com-

mittees. In contrast, the logic implemented by network management applications executes primarily at the central hosts of the NOC. These applications implement installation specific policies and are therefore customized to each particular network. Applications access the instrumentation and local state of the devices via standard management protocols. The management frameworks that are derived from these standards separate these applications from the the devices that they need to control. Because of their centralization and lack of extensibility, management applications suffer from several performance problems (see Section 3.5).

The management application processes in the platform hosts interact with a large number of rigid device servers. This interaction paradigm concentrates most processing into the platform workstations. The design of the management framework explicitly assumes that the network devices have limited computing resources for management purposes. Device servers are therefore relegated to collecting primitive data and making it available to the platform applications. These applications (1) retrieve data from the device servers using a management protocol (e.g., SNMP), (2) interpret and compute functions over the retrieved data (e.g., statistics), and (3) direct the device servers (e.g., how to handle fault scenarios). SNMP-agents are implemented as rigid, non-extensible device servers. Their MIBs and services are typically defined by a standards organization.

Example: The Micromanagement Problem

Let us consider a simple example of a network management task. Suppose that a network operating center needs to execute a management program to diagnose and handle port failures in a network hub. A skeleton code fragment of such a program is given in Figure 3.4. The symptom event evaluation function (`symptom_event()`), the management procedures (`diagnostic_test()`, `partition_port()`), and the managed variable (`port[]`), are all implemented within the hub itself. That is, these functions and variables are computed via instrumentation routines which are an integral part of the device. However, in a typical network management system, the program itself resides and executes at a host of the network operating center platform.

Let us consider the problems involved in accomplishing such simple handling of port failures using SNMP. The `symptom_event()` function describes potential port failures. This event is detected by polling some MIB variables that indicate the state of each port. Suppose that this event is signaled when the error-rate at a port exceeds a certain threshold. For instance, when 10% or more of the frames received on a given interface card are discarded due to errors. SNMP MIBs typically include error counters, (e.g., `ifInErrors`), that provide cumulative (integral) numbers of errors since the agent's boot-time. The values of these variables can be retrieved via SNMP's `Get` requests.

To evaluate the above event the management application needs to compute changes in error rates. That is, it needs to compute the second time derivative of the error counter variable. To perform this computation, the platform application

```
void
correct_problem() {
int sympttype;
   if (symptom_event(&sympttype)) {
       while(i=0;i<num_ports;i++)
             if diagnostic_test(port[i], sympttype)
                partition_port(port[i]);
       }
}
```

Figure 3.4: A Management Program Example

will need to poll `ifInErrors` at a very high frequency. Remote polling via SNMP introduces network delays in the observation of these changes. The mere detection of the symptom may be difficult or even impossible to accomplish via SNMP polling. We describe this problem in greater detail in Section 4.3.

Executing a `diagnostic_test()` requires (1) that the corresponding procedure has been predefined in the device server, and (2) that it can be invoked as a side-effect of a SNMP `Set` to a corresponding variable (e.g., `diagnosePorts`) on some private MIB. The platform application will then need to poll another MIB variable (e.g., `diagnoseComplete`) to ascertain that `diagnostic_test()` has completed its execution. Then it will need to reset this variable via another `Set`. It will then need to poll for the results of the test, in an MIB table whose entries record `port_failure` variables. Finally, for each positive `port_failure` it will invoke `partition_port()` as a side effect of another `Set`.

SNMP forces platform applications to micro-manage devices

This simple example shows how the SNMP interactions between NOC applications and device servers can force a platform host to micro-manage its network devices. *Micromanagement* occurs when a platform application must control the execution of a management program at the device by stepping the device's server through it. This occurs because the management protocol lacks appropriate expressive power to support more *"semantically rich"* interactions. For instance, SNMP does not support the composition of primitive management actions in a flexible and efficient fashion. Indeed, the limited primitive verbs of the management protocol lead to a very high rate of platform to device interactions. Manager platforms are forced to micromanage remote devices through the network, even for simple tasks. More complex management tasks require an even greater degree of interactions between the central hosts and the networked devices. Management applications are, therefore, seriously limited in processing non-trivial tasks.

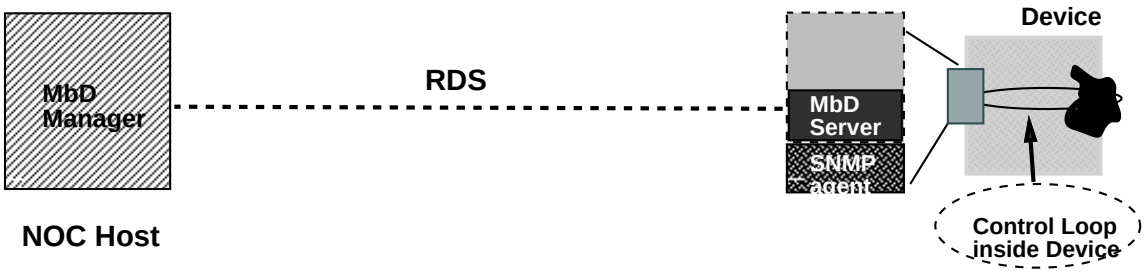

Figure 3.5: Control Loop inside the Device using MbD

3.1 The MbD Approach to Network Management

This dissertation introduces a novel approach to network and system management, namely *Management by Delegation* or MbD. While our main focus in terms of examples and applications is on network management applications, MbD is equally applicable to distributed system management applications[2]. The approach of MbD is to dynamically distribute the management computations to elastic servers (MbD-servers) at the devices where the managed resources are located. An MbD-server is an elastic process customized for network management and provides efficient bindings to management instrumentation, and support for SNMP interoperability.

For instance, consider the examples presented in Section 3.0.2. The management applications could delegate the computation of the diagnostics routines to the devices. By reducing the number of network interactions, the management application would gain much faster response time to the relevant events at the switch. Figure 3.5 shows how the "control loop" is made much smaller and kept inside the device.

MbD applies elastic processing to support spatial and temporal distribution of management functionality. Spatial distribution reduces the physical distance between an application and the devices that it manages. Spatial distribution reduces the overhead utilization of the network for management purposes and the delays involved in accessing remote data. For example, applications that need to evaluate and react to transient events of short duration, such as noice bursts in a line, need to be distributed to the devices.

Temporal or dynamic distribution is the ability to dynamically delegate new management code to a device when it is needed. Temporal distribution assists the developers of management applications to modify their management policies as administrative requirements change, and as the network environment grows and evolves. For instance, when a new pattern of failures is suspected at a switch, a new agent can be delegated to evaluate and correlate its symptoms. Management applications can use dynamic code delegation to address temporal problems, like the detection of

[2]See [Bauer *et al.*, 1993] for an analysis of the functional requirements of managing distributed systems.

intrusion attempts to a networked workstation.

3.1.1 Challenges of Distributing Management Functionality

The main problem that this chapter addresses is how to dynamically decentralize network management functions. There are several technical challenges in developing an effective distributed management environment. First, a decentralized management framework must enable applications to efficiently bind management code within the scope of each networked device. For example, delegated code must have direct access to the underlying instrumentation of the device.

Second, the new framework must be properly integrated within the standard management framework, so that existing applications and systems can interoperate with it. For example, management applications should be able to communicate and control the execution of delegated agents via SNMP. Third, a mechanism is needed to resolve non-deterministic management actions. For example, one manager may delegate an action (A) to reboot a device when certain failure is detected, while another independent manager delegated an action (B) to analyze the causes of the fault. If A is executed first, B will not be able to diagnose the fault.

3.1.2 Advantages of Management by Delegation

M_bD, in contrast with platform-centered management, supports a scalable model of management. Automation of management functions via delegated applications software reduces the load on operations centers. M_bD either eliminates or significantly reduces the need for intensive network polling, and therefore reduces the largest scale impediment of SNMP.

M_bD can greatly improve the autonomy and survivability of distributed systems. Management responsibilities may be delegated to devices to maximize their autonomy. For example, when communications are lost with the NOC managing processes, an M_bD-server may activate management programs that provide its device with fully autonomous management. Thus, an M_bD application can better handle the large management data rates involved during failures, and can directly access the control functions at the devices. Delegated agents can continue to execute while there is a network connectivity loss.

M_bD reflects performance tradeoffs driven by the realities of computing technologies. Networked devices are increasingly equipped with substantial computing hardware resources. These computing resources permit a degree of distributed management sophistication which exceeds (or even contradicts) the simple device models envisioned by the SNMP paradigm.

M_bD can inter-operate with current management protocols. It may also extend their capabilities, e.g., SNMP MIB entries may be flexibly programmable via delegated scripts. For instance, an M_bD-server can be used in conjunction with SNMP managers to solve the port-failure handling example of Section 3.0.2. A delegated diagnosis

program can generate an event notification and record its finding in an appropriate MIB table. A manager can then access the SNMP-agent incorporated within the M_bD-server to retrieve these MIB variables. We implemented a mechanism to provide concurrency control among concurrent management actions implemented as delegated agents. Each delegated agent can be instantiated with an attribute which represents a scheduling priority, and different agent scheduling algorithms can be configured as part of the M_bD-server.

3.2 Application Examples

Several management applications have been implemented using the M_bD model and software. In addition to some demonstration applications that we developed, groups at *Aerospace*, *Harvey Mudd College* [Erlinger *et al.*, 1994], *Smarts* [Dupuy, 1995], and *Synoptics*, have used M_bD code to implement several network management tools. This section presents three examples[3] of other network management application domains that can benefit from the functions supported by M_bD. Section 3.2.4 describes some of the common characteristics of management applications that require dynamic distribution.

3.2.1 Intrusion Detection

A *security threat* is the potential for deliberate (1) unauthorized access and manipulation of information, or (2) rendering of the system inoperable or unreliable. An *intrusion* is a successful set of actions to carry out a security threat. *Intrusion detection* is the ability of a computer system to automatically determine that a *security breach* has occurred. Anderson [Anderson, 1980] identifies three classes of malicious users that perform security breaches: (1) *masqueraders* are system penetrators that exploit a legitimate account, (2) *misfeasors* are legitimate users who participate in an illicit activity, and (3) *clandestines*, who seize supervisory control of the system.

Intrusion detection mechanisms assume that an attack consists of some number of detectable security-relevant system events. Surveillance and auditing facilities collect events such as remote logins and file access denials. Enormous amounts of audit data and computing resources are necessary for successful intrusion detection. Simply recording all of the audit records results in a huge amount of I/O and storage overhead. For example, if all audit events are enabled on a workstation, it can generate several megabytes of raw audit records per hour that must be read and analyzed.

Most intrusion detection systems use statistical analysis to measure variations in the volume and type of audit data. Summary statistics detect when the number of occurrences of an event surpass a reasonable threshold, which may indicate the presence of an intruder [Porras, 1992]. Profile based anomaly detection detects intrusions

[3]These examples were first presented in [Meyer *et al.*, 1995].

by monitoring audit logs for usage that deviates from standard patterns. Its main advantage is that it does not require a priori knowledge of the security flaws of any given system. However, there are several issues that hamper the effectiveness of such techniques, e.g., false positives and false negative rates. For instance, in some environments, e.g., academic departments, anomaly use is the norm among many users, producing a large rate of false positives.

Intrusion detection is an excellent candidate application for spatially decentralized and temporally distributed management. Centralized processing for intrusion detection won't scale up to large networked systems, and will require large amounts of network bandwidth to move audit data to a centralized point. If all the audit data is sent to a central location, it will result in significant network congestion. Correlation analysis must be performed on events in different machines' local logs. Since the computational requirements of intrusion detection scale in a worse than linear fashion [Meyer *et al.*, 1995], the audit processing must be kept distributed. Semantically rich conversations are needed between the distributed monitors, as they may need to pass relatively complicated structures that are hard to predefine in an MIB. Surveillance mechanisms must be adapted over time as new break-in modes are discovered.

3.2.2 Subnet Remote Monitoring

Consider the problem of evaluating the performance of a LAN segment. For example, a manager may need to determine which stations are generating the largest amount of traffic on the segment, in order to redistribute the load. On a busy network, such computations may require maintaining tables with entries for several thousand nodes, along with their packet counts. To download an entire station table to the central platform for processing is impractical. The entire transfer of the table could easily take a long time, even minutes. By the time the table transfer is completed, the performance data is likely to be obsolete and of little use.

A better approach is to distribute these computations to an M_bD-server located at a node in the corresponding LAN segment. Delegated agents may perform sorting and other processing locally, and provide summary and table information to remote managers via SNMP. For instance, they may compute the top n nodes sending packets, or an ordered list of all hosts according to the number of errors they sent over the last 24 hours.

For example, assume that a centralized application needs to evaluate the top n sending nodes of a given subnet. Assume that a sort will be performed based on the number of packets transmitted by each station. The management station would have to request statistics for all the hosts that have been seen on that subnet. The management platform will need to get a baseline count for each station and one to get the count for each station after a time t. The difference between the two sets of requests is then sorted by the platform to produce the top n list. If instead a decentralized approach is taken, the same function is delegated to an M_bD-server, and its aggregated costs are greatly decreased. Most or all of the computations happen

at the LAN, and are therefore more effective.

The RMON MIB [Erlinger, 1993] defines remote monitoring probes that collect information from LAN segments. An RMON probe is a device that collects performance data from a LAN segment and performs *predefined* calculations over that data. In the RMON MIB a limited spatial distribution of processing is used for the *Host Top-N* function. The Host Top-N MIB group provides sorted host statistics. A network administrator can define some of the parameters of the functions computed, e.g., the *data* items selected (e.g., top 30), and the *duration* (e.g., 24 hours) of the data collection. Thus, RMON provides a set of generic functions that can be remotely instantiated by network managers. Note that all the RMON computations have been pre-defined by the corresponding standards group. Thus, the RMON MIB does not support temporal distribution of computations over MIBs, as management applications can not define new types of calculations.

3.2.3 Controlling Stressed Networks

Network stress is loosely defined as the sustained operation of the network at "high" utilization levels. Stress may be caused by failure of network components and/or by high user demand. Characteristics of stressed networks include longer delays, reduced effective connectivity, increased packet traffic, unexpected routing, and unpredictable responses. Stress must be dealt with quickly, because it introduces instability that tends to escalate.

Networks in stressed conditions require different management strategies than unstressed networks. Management applications used for dealing with stressed network regions must have local autonomy, stress containment, domain stabilization, and gradual, graceful degradation [Meyer *et al.*, 1995]. Management algorithms should obtain most information locally at the devices, and only require low bandwidth to communicate outside of their domain. If the source of a problem is local, the local domain should be able to make decisions to contain and correct problems locally. Ideally, local domains should be able to anticipate stress conditions before they actually occur, and take appropriate countermeasures.

As network stress grows, management and network services should continue to function, albeit with worse performance. This requires a spatially distributed architecture, with few dependencies on remote resources. Stress monitoring involves the correlation of MIB variables (e.g., retransmissions, packet lengths, and timeouts) on a domain-by-domain basis. By conducting cross-correlations on a regular basis, patterns of stress could be discovered. Similarly, higher level managers would conduct cross-correlations of domain-manager information, to establish "regional" stress propagation. They could then devise and delegate the implementation of strategical and tactical policies to combat escalating stress. All these activities require management applications that are spatially and temporally distributed in a hierarchical fashion among network domains. We elaborate on the above problems in Chapter 4.

3.2.4 Centralized vs Distributed Management Applications

Different management applications require or admit different levels of distribution or centralization. An application that needs fast real-time decisions based on local device information will need decentralized control and intelligence. For instance, an application that keeps vital monitoring devices of a nuclear reactor should be able to perform its role even when the network is temporarily disconnected. Applications that utilize large amounts of data may need to perform decentralized processing to avoid overloading the platform and the network. In the other extreme, some applications must have a significant component of centralized intelligence and processing. For instance, consider an application that correlates global configuration about all the devices of a private network. Such an application can be more efficiently implemented in a centralized paradigm.

Polling Frequency

An example of an application that requires low levels of polling is a daily management backup control application. Such an application will retrieve every night a few MIB variables that ascertain if and when the last incremental backup was completed. Since these computations are based on slow-changing data, it is sufficient to use SNMP polling to collect the data. An example of an application that requires a high frequency of polling is a data compressing index function (see Section 4.5). This application depends on an ability to detect high frequency deltas on variables. The need for proximity to management data and the frequency of polling dictates that such computations be performed at the devices.

Management Information Throughput

This scale runs from a (1) high throughput/low information ratio to a (2) low throughput/high information ratio. Checking the liveness of a device connected via an 100 Mbps link via infrequent `ping` is an example of an application of type (1). Performing remote diagnosis of a large site via a 9600 bps modem link is an example of an application of type (2). Note that network throughput is affected not only by the amount of bandwidth available but also by the reliability of that bandwidth.

Semantically Rich Conversations

Some applications require semantically simple and infrequent conversations. For instance, an application may need to retrieve daily a few MIB variables for configuration backup purposes. Other applications need semantically rich and frequent conversations between manager applications and devices. For instance, an application may need to engage in a detailed diagnostic procedure every few minutes for some critical device. For instance, diagnosing failures in a controller device at a nuclear reactor may requires several hundred exchanges, following some expert system `if` rules. This is an example of a semantically rich and frequent conversation. The more

semantically rich the conversation is, the more it requires temporal distribution of management code.

Decentralized Applications

Decentralization is most appropriate for those applications that:

- have an inherent need for distributed control,
- may require frequent polling or computation of high frequency MIB deltas,
- include networks with throughput constraints,
- perform computations over large amounts of data, or
- have a need for semantically rich conversations between platform processes and device servers.

Centralized Applications

Not all management computations must be distributed. For instance, consider an accounting application that performs device inventories, e.g., counting the number of ATM interface cards in user workstations. Since these computations are based on slow-changing data, it is sufficient to use SNMP polling to collect the data. Other computations must be centralized. For example, consider an application that needs to find the least loaded ethernet segment to install a new device. To make this decision, the application needs to correlate performance data obtained from many segments. Another example is an application that displays an updated network connectivity map. Both examples are centralized because they need to merge and correlate information from several sources.

Spatial centralization is appropriate for management applications that have little inherent need for distributed control. Such applications (1) do not require frequent polling or high frequency computation of MIB deltas, (2) have high throughput bandwidth connecting the manager station and its devices, (3) exchange relatively small amounts of data, and (4) do not need frequent, semantically rich conversations between manager stations and device servers. Many network management applications in use are centralized, because the centralized (SNMP) paradigm is the only one that is ubiquitous. Therefore, most *first-generation* network management applications fit the above characteristics. The classic example of such first-generation application is the display of simple MIB variables, e.g., an MIB browser. Monitoring a router's interface status or a link's up/down status involves querying and displaying the value of a single or small number of (MIB) variables.

3.3 Design and Components of M_bD

The example of handling failures at a remote hub (in Section 3.0.2) illustrated some of the problems of performing a management task using current network management approaches. Let us describe how such a task may be implemented within the M_bD framework. First we need an elastic process at the remote hub, which is an M_bD-server that implements network management functions. This process allows delegated agents to directly monitor and control the resources of the managed element. For instance, the M_bD-server may support a function that evaluates events such as the `symptom_event` by monitoring the low-level hub registers associated with its ports. A management application may delegate an agent that dynamically binds with the existing management services code at the M_bD-server. Therefore, the entire management control loop happens inside the hub device. Such dynamic extensibility and control of the functions performed by the managed device reduces or even eliminates the micromanagement problem described earlier.

M_bD-servers maintain fast communications with the managed devices, often by sharing the same host computer with them. Therefore, the network access delays are replaced by bus or register access delays. An M_bD-manager is a management application process that delegates management agents (DPs) to the M_bD-server. These agents implement management tasks, i.e., they instruct the M_bD-server how to monitor and how to respond to what it observes. An M_bD-server can also be an M_bD-manager, i.e., it can delegate management functions to another M_bD-server.

A Sample Scenario of MbD

Figure 3.6 depicts an example scenario of a distributed management application that uses M_bD. An M_bD-manager, d, in host D is using RDS to transfer a DP to an M_bD-server in host A. For instance, this DP could be a program similar to that described in Figure 3.4. The M_bD-server is executing four agent threads which have been previously delegated, DPI_{1-4}. The DPIs communicate using the RDS_SendMsg and RDS_ReceiveMsg. An external client process, c, executing on host C, also uses RDS communication services to exchange data with the DPIs in the M_bD-server. DPI_2 is delegating a DP to the M_bD-server in host B. DPI_4 is accessing management services provided by the M_bD-server, For instance, it may be accessing a service to translate computed values into the required format for SNMP.

M_bD-server Roles

An M_bD-server can be a *proxy* SNMP-agent because it can access device data on behalf of remote management applications, and present it to them via SNMP. For example, the M_bD-server can collect diagnosis data on ports and save the observed data on an MIB. This feature is described in more detail in Section 3.4. An M_bD-server is also a *hierarchical mid-level manager* that executes management code on behalf of remote applications. An M_bD-server exports a dynamically changing set of services to

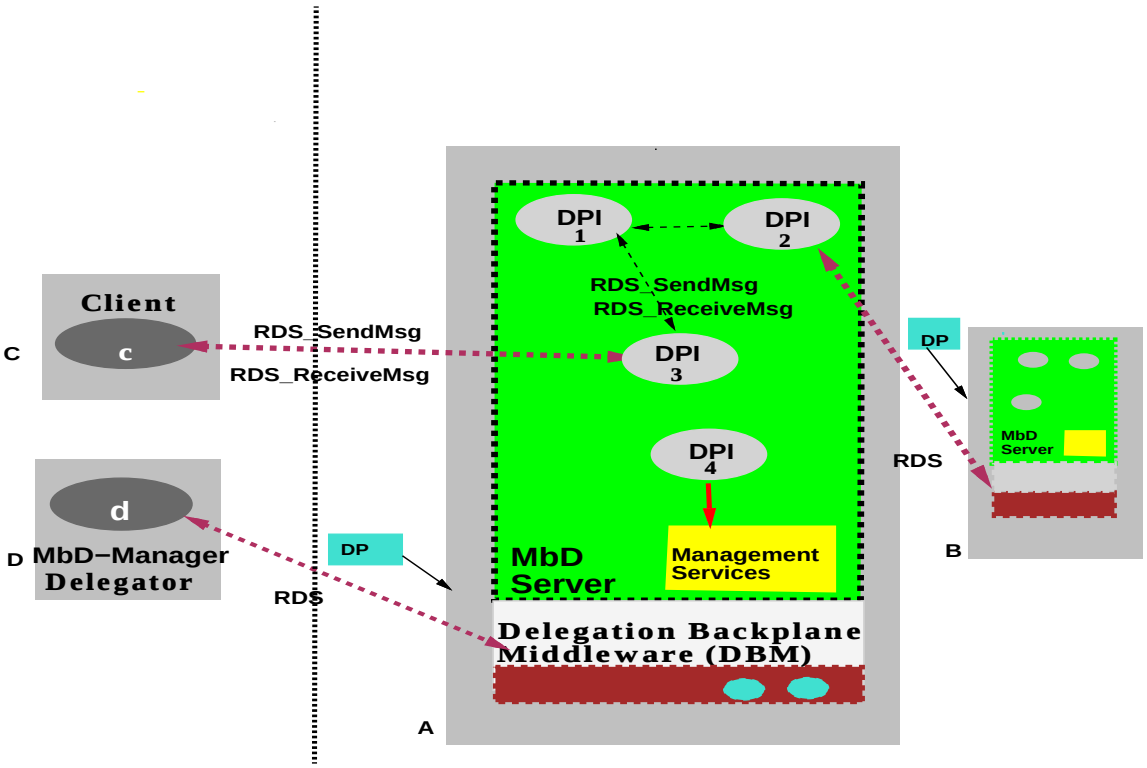

Figure 3.6: Delegation to an M_bD-server

external management processes. For example, the delegated management code in the above example can become a new service for remote applications. The M_bD-server also provides a configurable collection of management services for the DPIs that execute on its address space. For instance, these services may include local access to MIB data, access to SNMP services, device instrumentation routines, and so forth. An example of these services is given in Section 3.3.3.

3.3.1 Delegated Management Programs

A manager can delegate to a router device a DP to evaluate some statistic (e.g., the mean) of a MIB counter variable. From this DP, several DPIs can be instantiated to compute the same statistic function for different MIB variables, e.g., `ipInHdrErrors`, `ipInAddrErrors`, etc. The DPIs execute in the environment of the management data and functions that they need to access. The responsibility of computing these operators on management data is, therefore, dynamically shifted to the devices themselves.

A typical DPI will perform certain initialization steps, request services from the M_bD-server, and then wait for some event. For instance, a DPI will wait for instructions from a remote manager, or wait until a timer expires. A DPI will not poll

for messages, and will implicitly yield the CPU when waiting for a message. Most of their time is typically spent in a suspended state, resting between bursts of activity. They periodically collect information, analize it, and possibly take some action as a result.

For example, an NOC management application may need to know every time that n or more datagrams are discarded in a short period t due to errors in its IP header. A DPI may be instantiated to locally observe the corresponding MIB-II counter `ipInHdrErrors`. It will receive a message via `RDS_ReceiveMsg()` from a manager to indicate the value of the increment (n) and the time period (t). Whenever the value of `ipInHdrErrors` increments by n or more, the DPI will send a message to the corresponding manager via `RDS_SendMsg()`. Thus, a DPI takes the role of evaluating local events directly from the device, and forwarding to the manager application only the reports that it needs.

An M_bD-manager may control the execution of a DPI using RDS. For example, a manager may instantiate a DPI that computes statistical values of a given MIB counter variable. This DPI may send a report message to the manager process every time that it discovers an anomalous condition. A DPI may start sending more messages than its manager can handle at a given time, due to some abnormal condition. The M_bD-manager may then suspend and later resume this evaluation and reporting using the corresponding RDS services. Execution control via suspend/resume enables both the M_bD-server and the M_bD-manager to dynamically reallocate CPU cycles and bandwidth for urgent purposes.

3.3.2 Observation and Control Points (OCPs)

DPIs interact with *managed entities* via M_bD-server threads called *Observation and Control Points* (OCPs). An OCP provides an *Application Programming Interface* which is used by the delegated agents to interact with the managed entites. For example, consider an M_bD-server executing in a modem pool of an Internet service provider. Each modem is represented by an OCP that implements modem management services. For instance, a service such as `hangup()` to force the modem to drop a connection is implemented as a low-level modem-specific protocol that sets the internal registers of the modem. For different types of modes, the implementation of each service is likely to be different, but those details are hidden by the common service interface of the OCP.

In general, an OCP's service interface provides an encapsulation abstraction that hides many of the details of the entity-specific monitoring and control access protocol. OCPs may also be delegated agents, hence they provide a programmable interface to managed entities. An OCP can also provide an arbitrary concurrency control mechanism to regulate the access of delegated agents to the shared resources of a managed entity. For instance, the above OCP may serialize all the requests that involve setting the registers of the modem. OCPs provide granularity of access control, e.g., a given DPI can obtain bindings to a specific OCP but not to others. An OCP may

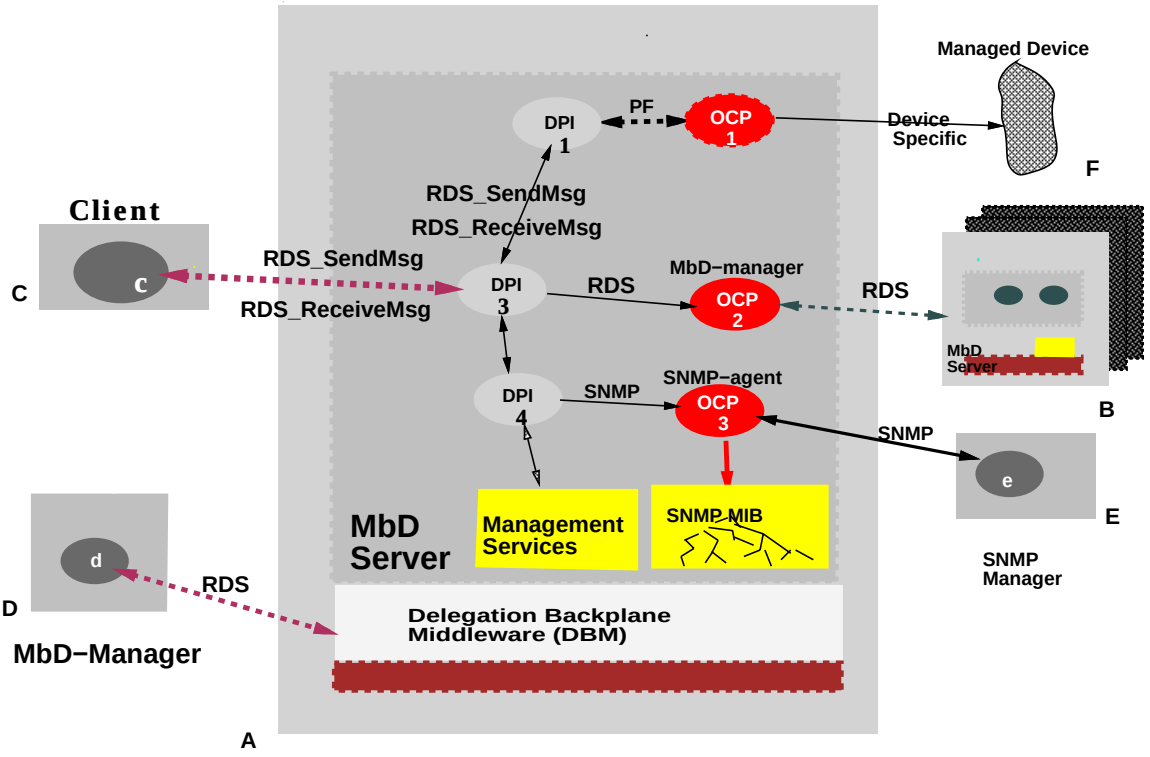

Figure 3.7: DPIs and OCPs in an M_bD-server

also perform services by invoking an external interface of managed object. Consider an M_bD-server at a file server computer. An OCP may monitor statistics of the TCP connections of the host by invoking an O/S specific command (e.g., `netstat`). The OCPs can locally store the values observed or computed, thereby caching frequently used information.

Figure 3.7 shows several delegated agents (DPIs) and OCPs executing inside an M_bD-server. OCP_1 provides an interface to the managed device F. It collects data from F using a device-specific protocol, and exports an interface called FP. OCP_2 is itself an M_bD-manager which has an RDS session with an M_bD-server at host B. OCP_3 is an SNMP-agent that is serving SNMP requests from (1) an SNMP manager, e, at host E, and (2) from DPI_4. Notice that this OCP supports an SNMP MIB. The other DPIs use RDS to communicate between themselves, with OCPs, and with remote processes.

3.3.3 Prototype Language and Services

The first M_bD-server prototype supports delegated agents written on ANSI C [ANSI, 1989]. C is ubiquitous, supports low-level facilities, and has efficient implementations. The prototype M_bD-server restricts the capability of a delegated management

mbd_SNMPclose()	mbd_SNMPgetCounter()	mbd_SNMPgetEnum()
mbd_SNMPgetInteger()	mbd_SNMPgetPort()	mbd_SNMPgetString()
mbd_SNMPgetVector()	mbd_SNMPopen()	mbd_SNMPsetCounter()
mbd_accept()	mbd_check_from_ent()	mbd_check_mesg()
mbd_check_mesg_from()	mbd_create_entity ()	mbd_delete_all_config()
mbd_entity_name()	mbd_entity_complete()	mbd_entity_status()
mbd_error_str()	mbd_exit()	mbd_get_from()
mbd_get_from_ent()	mbd_get_mesg()	mbd_get_ptr()
mbd_get_ptr_from()	mbd_id_for()	mbd_id_for_new()
mbd_init_dmp()	mbd_insert_config ()	mbd_io_id_for()
mbd_kill_entity()	mbd_lookup_ent_name()	mbd_my_entityID()
mbd_my_id()	mbd_name_for()	mbd_name_for_new()
mbd_new_entity()	mbd_parent_entity()	mbd_parent_id_new()
mbd_perror()	mbd_register_me()	mbd_remove_entity()
mbd_resume_entity()	mbd_send_mesg()	mbd_send_ptr()
mbd_send_ptr_to()	mbd_send_to()	mbd_send_to_ent()
mbd_set_ent_priority()	mbd_set_error ()	mbd_set_error_info ()
mbd_set_health ()	mbd_set_health_info ()	mbd_set_load ()
mbd_set_load_info ()	mbd_sock_close()	mbd_sock_readv()
mbd_sock_recv()	mbd_sock_recvfrom()	mbd_sock_recvmsg()
mbd_sock_send()	mbd_sock_sendmsg()	mbd_sock_sendto()
mbd_sock_writev()	mbd_sockbind()	mbd_sockbindlisten()
mbd_sockconn()	mbd_suspend_entity()	mbd_templ_name_for()
mbd_templ_name_for_new()	mbd_terminating()	mbd_thr_alive()
mbd_thr_change_quantum()	mbd_thr_count()	mbd_thr_dead()
mbd_thr_enumagt()	mbd_thr_enumerate()	mbd_thr_enumrecv()
mbd_thr_enumsend()	mbd_thr_getstate()	mbd_thr_instantiate()
mbd_thr_kill()	mbd_thr_libcset()	mbd_thr_resume()
mbd_thr_set_priority()	mbd_thr_sleep()	mbd_thr_suspend()
mbd_thr_status()	mbd_thr_status_new()	

Table 3.1: Prototype M_bD-server allowed functions

agent to invoke arbitrary external functions. They may only invoke external functions that are made explicitly available in a configuration file. A sample set of these functions is presented in Table 3.1. This list defines an API for delegated agents to request services from the M_bD-server. The API includes functions to control the scheduling execution of threads, exchange messages with other agents, OCPs, and M_bD-managers; and receive notifications of events. Notice that this list is a configuration option for each M_bD-server, that is each incarnation of an M_bD-server can have a different list, and the list can be changed during execution.

3.3.4 Controlling the Execution of Management Agents

Consider an M_bD-server to whom two management tasks have been delegated by two different managers. Assume that both of these agents share the same *"triggering"* event E, i.e., they are waiting on the same event. When E occurs, these management actions are executed, and their execution order may result in substantially different results.

Example: Concurrent Management Actions

Consider an X.25 controller within a switch. The controller has an M_bD-server responsible for the controller operations, which are mostly *Virtual Circuits* (VC) operations. Management applications have delegated three agents, D_1, D_2, and D_3, to the M_bD-server. All D_i are to be triggered upon the event E = *buffers-full* (i.e., all VC buffers are full):

- D_1 is delegated by a vendor-provided switch manager, and specifies the following: run local diagnostics to check possible link or controller failures, then reboot the controller upon fault.
- D_2 is a congestion handler delegated by the network control center that specifies: evaluate and report performance parameters, abort some VCs, and set limits on local resources usage.
- D_3 is a flow-control action pursued by network flow-control protocol acting in a manager role. It specifies: dump buffers contents and reset flow-control on VCs.

This example illustrates typical forms in which non-deterministic management actions distributed through networked systems can lead to unpredictable results. This unpredictability can be attributed to three factors: (1) the arbitrary order of the execution of actions, (2) the division of work between local and remote management functions, and (3) the interactions between them. The arbitrary order of execution of several actions that are triggered by common events introduces management problems.

In the above example, any order of execution pursued in applying the three actions will lead to very different behaviors. If D_1 is executed first, the controller may be rebooted. When D_2 then evaluates the performance parameters it would sample and report to the global manager values that report the state of the device after re-initialization. These are likely to be different values than those that would be reported by executing D_2 first and then D_1. As a result the controller may pursue unobservable intermittent failures as it reboots and fails again. Global management applications which expect reporting through D_2 will never be notified about the problem.

Concurrency Control in MbD

We have implemented a mechanism to provide concurrency control among delegated agents. Each agent can be instantiated with a numeric attribute which represents a priority property. For instance, different priorities can be associated with different management applications, or management stations. These properties are used by the DBM scheduler to resolve execution conflicts in a deterministic fashion. The M_bD-server attaches this priority attribute to the internal representation of each delegated agent.

The original DBM Scheduler (described in Section 2.3.4) implements a preemptive round robin scheduler. Two additional preemptive scheduling policies were implemented to control nondeterminism of management actions: *Wait_Wound* and *Wound_Die*. Wait_Wound is non-exclusive. Delegated agents with lower priority will wait for those with higher priority before being scheduled. Agents of the same priority will execute concurrently. Wound_Die is exclusive. Only one delegated agent can access a managed object at the same time. Agents with lower priority will be terminated when they are in conflict with a higher priority agent.

In the above example, scheduling priorities could be associated with the different manager entities. For example, the switch manager has the lowest priority, the flow control center has a slightly higher priority, and the control center has the highest priority. Thus Priority(D_1) < Priority(D_3) < Priority(D_2),

3.4 MbD Integration with SNMP

3.4.1 Extending an SNMP-agent

M_bD can inter-operate with and extend current network management protocols, such as SNMP. This integration allows standard management applications to benefit indirectly from M_bD's extended functionality. Figure 3.8(a) shows the structure of a typical SNMP-agent. It includes an external interface that supports the SNMP protocol, the implementation of an MIB, and the instrumentation to obtain data from a real device. Figure 3.8(b) shows an M_bD-server that augments the equivalent SNMP-agent. For example, an M_bD_{SNMP} process may implement an SNMP-agent as an OCP thread. Using the M_bD_{SNMP}, a manager process can dynamically delegate agents that

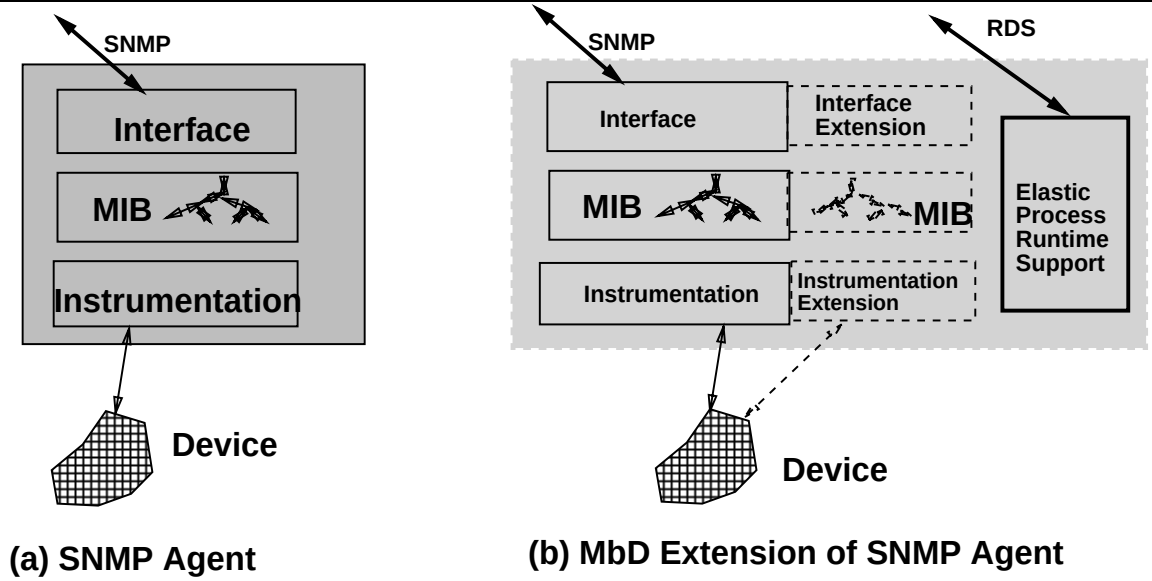

Figure 3.8: Extending an SNMP-agent

implement new interfaces, new MIB extensions, and new instrumentation functions. The following examples illustrate the benefits of integrating M$_b$D with SNMP.

3.4.2 Delegating via SNMP

An SNMP manager process can use SNMP requests to delegate an agent to the M$_b$D$_{SNMP}$. The SNMP manager invokes `GetNext(private.mydmp.location.row)`, to obtain a reserved MIB table row. The M$_b$D$_{SNMP}$ will then allocate the row and return to the manager the row index, e.g., `1234`. The manager will then use this table row index to identify the delegated agent in the M$_b$D$_{SNMP}$. For instance, the manager will invoke
`Set(private.mydmp.location.1234, ''grande.cs.columbia.edu:/tmp/mydmp.c'')`.
The M$_b$D$_{SNMP}$ will interpret this command as a request to delegate the corresponding agent code (`/tmp/mydmp.c`). The M$_b$D$_{SNMP}$ will then use a file transfer protocol (e.g., `tftp` [Sollins, 1992]) to retrieve the agent code from its location (`grande.cs.columbia.edu`).

Example: A Delegated Management Agent

Consider a management task that (1) detects a symptom of a fault, (2) performs some test to evaluate its type, (3) notifies the NOC manager console, and (4) executes a preventive action, e.g., partitions the ports that have failed. A skeleton program that performs this task (shown earlier in Section 3.0.2) is given in Figure 3.9. The entire procedure can be delegated at boot time to an M$_b$D-server that executes at the device. An agent thread can be instantiated via an SNMP `Set` from a simple SNMP application. For example, in the above example, an SNMP

```
void
correct_problem() {
int sympttype, i;
   if (symptom_event(&sympttype) == PROBLEM) {
       while(i=0;i<num_ports;i++)
             if diagnostic_test(port[i], sympttype)
                partition_port(port[i]);
       }
}
```

Figure 3.9: A Delegated Management Agent

manager will use a `Set` command on a variable of the corresponding row, e.g., (`private.mydmp.location.instantiate.1234`) to create a new instance of the delegated agent. The device will then assume autonomous responsibilities to handle port failures. Communications between the platform and the device occur during delegation time and at agent instantiation, but not a single data exchange is required during stress time.

Example: MbD can Support MIB Filtering

An M_bD-manager may delegate an agent that implements a *filtering* algorithm over an SNMP MIB. This agent can use local services to store the filtered values in an MIB table. This table can then be accessed by remote SNMP-managers, using `Get-Next` to retrieve an ordered collection of filtered values. Filtering is one of the distinctive capabilities of CMIP (see Section A.5) which is not present in SNMP. In CMIP, a filter is a Boolean expression consisting of assertions (equality, ordering, presence, or set comparison) of attributes of managed objects. M_bD extends SNMP to support data filtering using arbitrary functions. An extended example of filtering is given in Section 5.3.1.

Agents can interact via SNMP

Consider, again, the previous example. Let us assume that policy constraints require that a human operator control the decisions to partition the ports of a hub from an SNMP console. This requires a split of the original program into two delegated agents. The first one will generate specific SNMP `Traps` to the SNMP console, and is shown in Figure 3.10. The notify() operation is implemented as an SNMP trap that informs the SNMP console of the current condition. These traps can then be used to invoke appropriate `Gets` of `port_failure` table entries, indicating which ports are in trouble. Operators can then make the decisions concerning the partitioning of ports

```
void
local_correct_problem() {
int sympttype, i;
   if (symptom_event(&sympttype) == PROBLEM)
        notify(manager_id, sympttpe);
}
```

Figure 3.10: A fault-detection/notification program

by setting an MIB variable that represents the `partition_port` procedure. Setting this MIB variable to j will cause the invocation of `partition_port(j)`. Chapter 5 presents several examples of interoperation between SNMP managers and M$_b$D applications.

3.4.3 M$_b$D Interoperability

The same integration that we demonstrated for SNMP could be done for other management protocols, e.g., CMIP. Therefore, M$_b$D enables network management systems to effectively extend the capabilities of their current frameworks. Contrast this flexibility with current MIBs which only provide rigidly predefined data. M$_b$D allows management applications to obtain the data that they are really interested in. We elaborate on this subject in Section 4.3. For example, the RMON MIB permits remote configuration of pre-programmed monitoring processes. M$_b$D permits delegation of monitoring processes dynamically and the recording of the values which they compute within appropriate MIB variables. For instance, M$_b$D enables *programmable* versions of monitoring variables, such as those predefined in the RMON MIB.

3.5 MbD vs Centralized Management Approaches

This section compares the network management paradigms of M$_b$D and those of centralized standard approaches. For illustration purposes, SNMP is used as a representative network management protocol. The discussion, however, is equally valid for CMIP, since these limitations result from the very paradigm that they share rather than from design features specific to each protocol. The section examines some of the performance problems which are intrinsic to network management applications. We compare the performance of an application using SNMP and M$_b$D. We compare SNMP and M$_b$D with respect to several problems related to scalability, reliability, resource constraints, and semantic heterogeneity. These problems are illustrated by examples that are germane to the realm of network operations and system management. However, note that these problems are instances of generic problems that apply to many other types of distributed systems and applications.

```
void
diagnose_and_partition() {
int i;
   while(i=0;i<num_ports;i++)
        if (SNMP_Get(port.diagnostic[i]) == FAILURE)
          SNMP_Set(disconnect_port[i]);
}
```

Figure 3.11: A Management Program Example

3.5.1 Performance

We illustrate the performance characteristics of M$_b$D by comparing the performance of a management application using SNMP to the performance of the same application using M$_b$D. Consider the problem of diagnosing port failures in a network hub, and disconnecting malfunctioning ports.

Figure 3.11 presents a skeleton implementation of a centralized port diagnostic procedure. Every invocation of `diagnose_and_partition` requires a minimum of $n =$ `num_of_ports` SNMP `Get` requests. Let us assume that m of the ports are diagnosed as failures. The program therefore needs m `Set` requests for the disconnection actions. For simplicity, let us assume that all SNMP messages have the same delay. The overall response time cost function for this SNMP interaction can be approximated by

$$T_{\text{SNMP}} = (n+m) * (t_N(\text{SNMP})) + t_{mgr} + t_{\text{MIB}}.$$

Here $t_N(\text{SNMP})$ represents the network delay of each SNMP message, t_{mgr} represents the local computational time of the manager process at the NOC host, and t_{MIB} represents the time spent by the SNMP-agent MIB at the hub. The cost of each SNMP request depends on the following factors:
(1) The time required to generate an SNMP request message in ASN.1.
(2) The delay of the request message from the manager to the SNMP-agent.
(3) The time required by the SNMP-agent to generate a response message in ASN.1.
(4) The delay of the request message from the SNMP-agent.
(5) The time required by the SNMP-manager to parse the response message.

Consider an M$_b$D implementation of the same example. The code of Figure 3.11 becomes an agent that is delegated to an M$_b$D-server executing at the hub, with cost $T_{delegate}$. Now the invocations of the SNMP `Get` and `Set` actions are bound to *local* routines that can access the SNMP MIB. An invocation of `diagnose_and_partition()` requires now only one network interaction. After delegation, the typical response time cost function for this M$_b$D interaction can be approximated by

$$T_{MbD-request} = t_N(request) + (n+m) * t_{local}(\text{OCP}) + t_{MIB}.$$

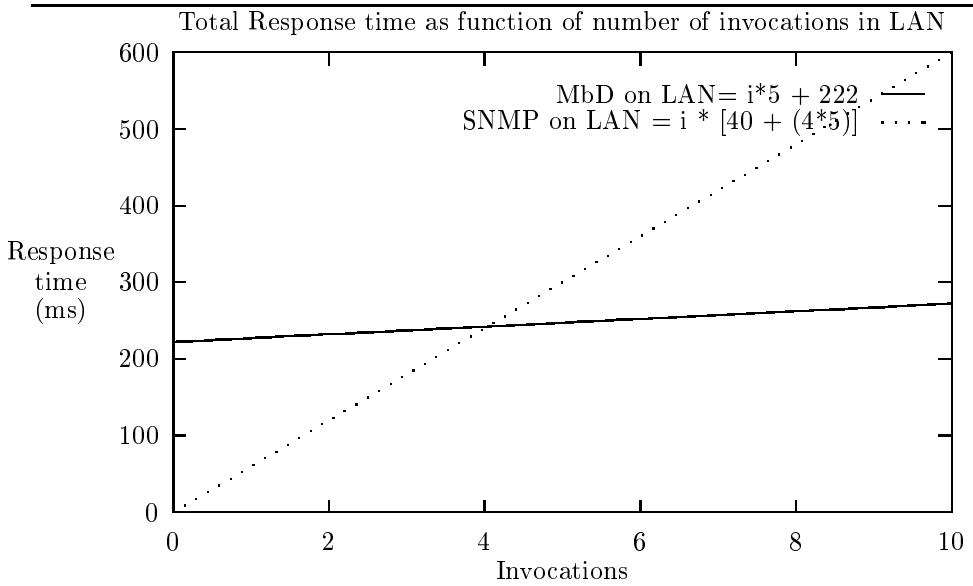

Figure 3.12: SNMP vs M$_b$D on a LAN

In this formula, $t_N(request)$ is the invocation request, which is very likely to be smaller than $t_N(\text{SNMP})$. $(n+m) * t_{local}(\text{OCP})$ represents the $n+m$ local requests to diagnose and correct the ports. We have now moved the network-polling of the diagnostics to be a local computation on an OCP executing at the hub device. Thus, the delay associated with accessing the MIB is very small, i.e., $t_{local}(\text{OCP}) << t_N(\text{SNMP})$. Of course, we still need to amortize the initial cost of delegating the agent,

$$T_{delegate} = t_N(delegation) + t_{MbD}(integration).$$

Here, $t_N(delegation)$ is the time required to transfer the delegated agent code, and $t_{MbD}(integration)$ is the time required for compiling and linking it into the M$_b$D-server. The residual performance difference after delegation is

$$T_{residual} = T_{\text{SNMP}} - T_{MbD} = (n - 1 + m) * (t_N(\text{SNMP})) + \delta.$$

For instance, if there are $n = 16$ ports in the hub, and only one has a problem, $m = 1$, $T_{residual}$ is equivalent to 16 SNMP requests, and a very small δ.

Performance Evaluation

Let us assume that there are 32 ports in the hub ($n = 32$), and that none of them are having a fault ($m = 0$). Let us assume the following values (in ms) for the SNMP cost equation $t_{\text{MIB}} = 30$ and $t_{mgr} = 10$. Each SNMP message ($t_N(\text{SNMP})$) adds a delay of 5 ms on a LAN and 100 ms on a WAN. An SNMP application could combine several requests into one SNMP message, so we assume that 8 requests can fit in one

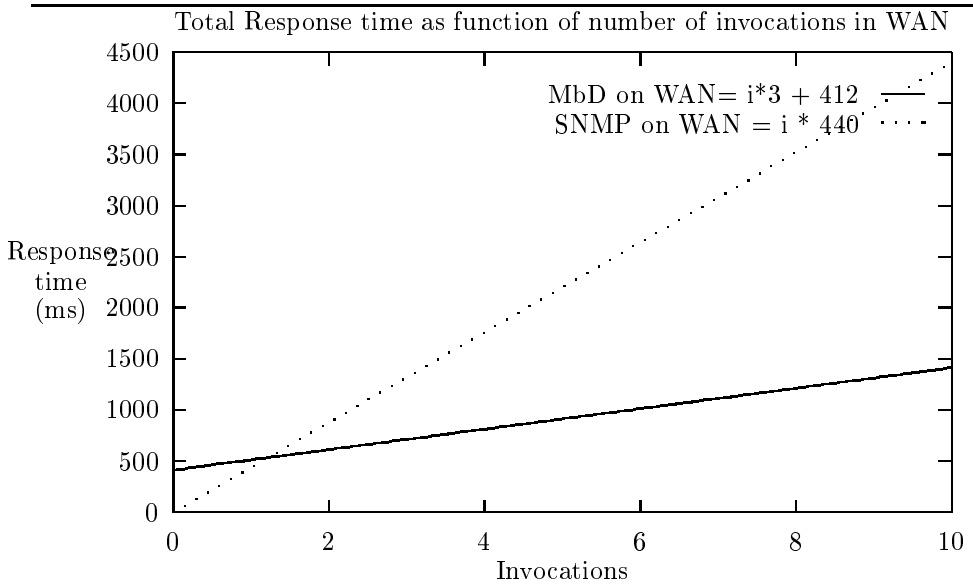

Figure 3.13: SNMP vs M_bD on a WAN

SNMP message. Then the total time required for one program invocation using SNMP is

$$T_{\text{SNMP}} = (32/8) * t_N(\text{SNMP}) + 40 = 4 * t_N(\text{SNMP}) + 40,$$

and for i invocations,

$$T_{\text{SNMP}}(i) = i * [4 * t_N(\text{SNMP}) + 40].$$

The total time required for one program invocation using M_bD is $T_{MbD} = T_{delegate} + T_{MbD-request}$. Let us assume the following values for it: $t_N(delegation)$ is 10 ms on a LAN and 200 on a WAN. $t_N(request)$ is 5 ms on a LAN and 100 ms on a WAN. $t_{MbD}(integration)$ is 100 ms, and $t_{local}(\text{OCP}) = 1ms$. Then,

$$T_{MbD} = t_N(delegation) + 100 + t_N(request) + 32 + 30$$

and for i invocations,

$$T_{MbD}(i) = i * [t_N(request)] + t_N(delegation) + 162$$

The actual performance of such an application is influenced by system parameters, such as the speeds of the CPUs of the platform and device hosts and the network bandwidth and delay. For simplicity, let us assume that the SNMP-manager and the M_bD-server hosts are equivalent in CPU performance. In addition, the average load on these resources should be taken into account. Figure 3.12 illustrates the performance comparison between M_bD and SNMP in a LAN and Figure 3.13 in a WAN. The differences in total delay are larger for a WAN, because of the increased latency. In a WAN it takes 2 invocations of the program to fully amortize the initial cost of delegation, while in a LAN it takes 5. Of course, this amortization will be much faster for applications that require larger amounts of device data.

3.5.2 Scalability

Platform-centric network management systems do not scale up to large networks. An SNMP manager process may interact with a large number of SNMP-agents. Each SNMP interaction involves retrieving and analyzing MIB data. This interaction pattern has two characteristics: (1) it concentrates most processing into the manager's host computer, and (2) it entails a high degree of communications involving the manager's host. Because of this centralized interaction pattern, SNMP establishes implicit scale limitations. A networked system becomes unmanageable when there is an increase in the number of managed devices or when there is an increase in the number of managed variables that their MIBs support.

Example: SNMP Polling

Consider an SNMP application executing on a dedicated management station that manages a set of point-of-sale registers connected by a LAN. The management station polls each register via SNMP to retrieve its operational state. There is a limit on the maximum number of variables that can be polled and the frequency of polling. The maximum number of registers that the management station can handle is bound by the length of the polling interval divided by the time required for a single poll request. For supermarket point-of-sale registers, a reasonable polling frequency is every 10 seconds [Eckerson, 1992]. The maximum tolerable delay will be much lower for many real-time applications (e.g., nuclear reactors), and higher for others (e.g., printers). In a Wide Area Network, larger delays will make the number of SNMP devices that can be queried an order of magnitude lower. A similar example is described in more detail in [Ben-Artzi *et al.*, 1990].

Example: Moving Large Tables

Consider a future ATM switch providing services to several thousand video-on-demand subscribers. The network management system must keep large tables of ATM entities that need to be processed from time to time. For instance, each port would have an entry in the corresponding MIB table for configuration changes. The NOC platform hosts will typically be located at a significant distance from the switches. Retrieving such large tables using SNMP is very inefficient. Moving the entire table to the NOC hosts to search for a few ports is very wasteful of both network bandwidth and platform host CPU cycles. Because of the platform-centric paradigm, data analysis is conducted only at the NOC. Thus, it requires data access and processing rates that do not scale up for large and complex networks. Platform applications may be unable to sustain the required level of communications to maintain an accurate representation of the real-time MIB information. Nor will they be able to afford the computing cycles required for data processing and presentation in the management station.

Human intervention

Scalability is even more seriously bound by the rates at which operators can handle data and alerts. As more data is collected and processed, more failure symptoms are detected and displayed on consoles. Operators' consoles become flooded with unimportant and/or redundant information. Many management applications produce alerts for trivial events. For instance, when users go home for the day and turn off their desktop computers, red lights go off on management consoles. Important event notifications (e.g., a file system is down) are often delayed and temporarily lost in a sea of alerts at the NOC. The task of the operators that interpret these symptoms becomes increasingly more taxing as the network grows. Furthermore, *"A critical problem facing operations managers today is the scarcity of trained personnel"* [4]. If the data collected could be interpreted by software applications instead of human operators, this problem would be simplified. Platform-centered management, however, establishes significant barriers on the development of such applications.

MbD is Scalable

Instead of bringing data from the devices to platform-based applications, an M_bD-manager delegates parts of the management applications to the devices. For example, an operations management center can delegate management functions to an M_bD-server at a switch, programming it to execute certain tests at a given time, or upon discovering a given event. An NOC host is relieved from polling by delegating agents to the network switches to monitor and detect network failures. The switches can invoke other delegated programs to isolate and handle a specific failure, without unnecessarily involving the delegator.

Automation of management functions via delegated applications software reduces the load on operations centers. Since the management data is interpreted at the M_bD-server, the monitoring demands on human operators are significantly reduced. M_bD applications can flexibly control the granularity of their platform-device interactions, and thus they can avoid micro-management. M_bD supports the composition of primitive management actions in a flexible and efficient fashion. In summary, M_bD either eliminates or significantly reduces the need for intensive network polling, and therefore reduces the largest scale impediment of SNMP.

3.5.3 Management Failures and Reliability

Consider a network failure that involves several LANs. During such a failure, centralized SNMP applications will tend to increase the rates of data access at the remote devices. This will happen at a time when the network is least capable of handling the excess load. The data collection rates required by the applications may easily exceed the bandwidth available. A bandwidth-demanding SNMP application

[4] Rick Sturm, Communications Week, October 23, 1995.

may be unable to perform under such limitations. The rates at which management data must be tracked by the platform may exceed its processing capacity.

The time scale over which the behaviors of such networks need to be monitored, and management actions invoked, is too short. The centralized control inherent in SNMP can cause network instability, due to the time length of the control loop. A control loop starts at the time that an event occurs at the device, continues when the central application realizes that there is a problem, and completes when the device receives instructions to handle the problem. Such feedback loops will tend to oscillate more severely as the feedback time increases.

The platform-centered approach is, therefore, significantly limited in its ability to handle the failures arising in large-scale internets. An SNMP application could neither handle the enormous data rates involved, nor access the control functions at the devices fast enough. The average management response-time will tend to stretch. The likelihood of management failures will increase, due to delay or even loss of management communications, at a time when fast and reliable response is most needed. Management applications will perform worst when they need to perform best.

Platform-centric management produces failure-prone communication bottlenecks. Since the platform host contains most management functions, it is rendered most vulnerable to network failures. Even simple failures could load the managers' bandwidth and cycles, potentially bringing it down. If the platform host is down or overloaded, devices cannot accomplish recovery, as they must wait for instructions from the platform application. Thus, even a minor problem may potentially lead to an avalanche failure of the entire network management system.

Some networks may often experience both short-term and extended periods of interrupted connectivity between the management platform and managed devices. The central-platform paradigm becomes almost useless during such periods. Under stress conditions, platform-centered management requires increased access to device agents in order to handle failures. Such increased accesses may worsen the network stress and accelerate failures. Decreasing network reliability will also cause loss and delay of platform-agents interactions, leading to potential failures of the network management system.

MbD Improves the Autonomy and Reliability of Managed Devices

M_bD can greatly improve the autonomy and survivability of distributed systems. As the network grows, or involves more complex devices, management responsibilities may be delegated to devices to maximize their autonomy. Devices may acquire autonomous management capabilities, conditional on the network status. A device can use the status of its environment to instantiate appropriate management programs, reflecting different levels of autonomy. For example, when communications are lost with the NOC managing processes, an M_bD-server may activate management programs that provide its device with fully autonomous management.

An M_bD application can better handle the large management data rates involved during failures, and can directly access the control functions at the devices. Delegated agents can continue to execute while there is a network connectivity loss. In networks where management communications bandwidth is scarce, device autonomy can be increased. In high-speed networks where fast reaction is needed, an M_bD-server responsible for such control functions may be located in close proximity to the managed elements. Delegation and instantiation of agents may be conveniently scheduled (e.g., for device initialization time) leaving only minimal communications for stress times.

3.5.4 Resource Constraints

The type and quantity of computational resources available for management purposes varies greatly among networked devices, depending on the limitations of each device's hardware. Small devices, like modems, will typically offer very limited computational capabilities for management purposes. Mobile devices may have limited computing resources due to their power consumption and storage limitations. In contrast, a telecommunications switch or a file server are more likely to allocate much larger computing resources for management. Large, general-purpose devices like workstations computers can invest more substantial computing resources to support extensive manageability.

Administrative Restrictions

Administrative policies may impose other restrictions in the allocation of management resources. For instance, a security policy may prescribe the use of strong encryption on devices located at specific sites. Also, the type and amount of resources available for management of a given device may vary over time. For instance, a telecommunications switch may allow a larger amount of memory for management tasks at off-peak hours. Any device may provide a larger amount of resources for brief periods of time for specific purposes. For example, a host being exposed to a penetration attack will allocate resources in order to track down the intruder.

Device Capabilities

Current management standards lack any effective mechanism to differentiate between the management capabilities and resources of different devices. The variation of management capabilities among various devices creates conflicting interests between different constituencies. For instance, whenever a new management feature is proposed, there are many types of devices which cannot afford to allocate sufficient resources to support it. This conflict has lead the network management community to many non-conducive debates over the feasibility and relative cost of any proposed management feature. Frequently, the only common ground found for standardization is at the lowest possible common denominator. In other words, network management

standards tend to underestimate the resources available at managed elements. Because of this, inexpensive resources embedded in network elements are not effectively used for management.

Some network devices commit substantial processing resources to management. In some cases, they may exceed the proportional share of resources available to them at the NOC hosts. Thus, management data is often moved from an embedded device through the network to be processed at a platform host where it must compete for lesser resources. For example, a Synoptics hub concentrator is equipped with a dedicated SPARC CPU for management purposes. If a network device is equipped with significant computing resources, it is more efficient to allocate a fraction of these resources to process the data locally rather than to move it. Platform applications occasionally need some of this data, for instance, to correlate observations from multiple devices. But it is more often the case that the platform merely retrieves the data from a given device and processes it by device-specific applications. For instance, an application would retrieve private MIB variables from a Synoptics hub, and then use a Synoptics specific application at the NOC host to evaluate the data.

Resource Constraints in MbD

M_bD applications can be tailored to the computational resources available at each networked device and at particular times. An M_bD-server for a resource-constrained device can be configured to accept only certain types of delegated agents, i.e., those that take into consideration resource constraints. For instance, the M_bD-server of a modem would only allow delegated agents that can monitor the modem's registers, and maybe perform some arithmetic computations with them. M_bD applications can allocate management functions to the devices according to current administrative policies. For instance, delegated agents can be used to execute security log analysis at off hours.

Performance Tradeoffs

Networked devices are increasingly equipped with substantial hardware resources (e.g., hubs incorporate powerful RISC CPUs). These computing resources permit a degree of distributed management sophistication which exceeds (or even contradicts) the simple device models envisioned by the SNMP paradigm. The costs associated with NOC operations dwarf those involved in procuring additional hardware resources within devices. Therefore, it is cost effective to procure devices with sufficient resources for distributed management. These technological and economic factors contribute to dramatic changes in the nature of emerging and future networks. In such networks, a distributed, device-oriented management paradigm is more flexible, scalable, and less expensive than the centralized, static management paradigm represented by SNMP.

3.5.5 Critical Evaluation of Standards Management Models

While standardization resulted in significant advances in network management, manageability requires more than means to query network devices. The standard management frameworks are essentially "data-movers". The standards define protocols to *move* data around, instead of defining how to *process* the data. Management data travels from a source to a destination, and little gets done with it.

Current management systems pursue a *platform-centered* paradigm. This paradigm separates management applications, logically and physically, from the data and services that they need. Platform-centric frameworks allocate almost all responsibilities to the manager applications executing at the platform. Distributed management applications are implemented following a traditional Client/Server (C/S) process interaction paradigm. The C/S paradigm associates functionality with device servers rigidly. A client process in a manager role can only invoke a *fixed* set of predefined services. These services cannot be modified or expanded without the recompilation, reinstallation, and reinstantiation of the server process.

For example, SNMP-agents are device-based servers that perform only menial tasks, e.g., collecting device-related data. The services provided by any SNMP-agent have been strictly defined by standards. Their MIBs have been rigidly defined at their design time. SNMP-agents collect device management data and provide a query interface for remote manager applications. The platform-centered paradigm, therefore, introduces a wedge that separates these applications from the the devices that they need to control. This rigid division of functionality hinders the development of effective management systems.

The explicit assumption of the SNMP framework is that network elements can not afford the computing resources needed to be "intelligent". Thus, they must rely on the centralized smarts of the platform-based applications. This paradigm assumes a primitive networked environment, where devices lack resources to execute non-trivial management software. Since devices have limited computing resources disposable for management purposes, their servers should be very simple and perform only minimal duties.

In the days of glass-house environments, network devices were indeed resource poor. Also, management data and functions were relatively simple, and organizations could devote sufficient personnel to handle operations. Within modern heterogeneous networked systems, however, many (or most) new devices incorporate significant and ever growing processing resources. At the same time, management data and functions are much more complex, and the human resources needed for operations are scarce and expensive.

Over time, the cost of device equipment will continue to decline as it is driven down by a commodity market. In contrast, the additional cost of operations are driven up by growth, distribution, and heterogeneity. The circumstances, premises, and problems associated with manageability in such environments are significantly different than those envisioned by the platform-centric approach.

The above management models establish barriers to effective network and

system management of modern distributed environments. M$_b$D seeks to address these limitations, and is orthogonal to the choice of methods to collect, organize, or access managed data at agents.

A Comparison of M$_b$D and SNMP

Table 3.2 summarizes the differences between the SNMP and M$_b$D paradigms.

Model	M$_b$D	SNMP
Function Allocation by	Dynamic distribution Application developers	Static Definition Standards bodies
Analysis and Control at	Spatial Distribution devices	Centralized platform
Devices are resources are	intelligent abundant	dumb limited
Management traffic overhead	minimal	high - polling
Network failures continue operation	autonomous devices yes	devices need platform limited
MIB definition	Dynamically extensible	Fixed and predefined
Interaction platform-device	fine grain, adjustable	micro-management
Device instrumentation	local access	via network
Real time constraints	device bound	network bound

Table 3.2: M$_b$D vs SNMP Comparison

3.6 Conclusions

M$_b$D provides a simple model to dynamically compose management systems, by connecting and integrating independent delegated agents. Designers and vendors of network devices can provide libraries of predefined management routines that can be used to compose these management systems. Network managers can use these libraries and their own programs to build distributed multi-process management applications. An M$_b$D-server provides a flexible way for network managers to execute, configure, and control this open-ended set of management programs in close proximity to the managed devices. By reducing the length of the control loop, management applications gain much faster response time and reliability. M$_b$D is an effective tool to dynamically allocate management tasks responding to the vicissitudes of networked environments.

4

Evaluating the Behavior of Managed Networks

4.1 Introduction

The global behavior of a network domain is defined by the aggregation of the behaviors of all its element components. Large network domains can have tens of thousands of elements which interact via many protocols (e.g., TCP/IP, SNA, NetBios), over different connection fabrics (e.g., Ethernet, Token Ring, FDDI). The behavior of each network element is characterized by a large number of local indicators, some of which can be found in their MIBs. A typical router MIB, for instance, can have several thousand variables representing its routing tables, interfaces, and so forth. Many behavior indicators are functions over these MIB variables that are computed by management applications. For instance, a network utilization function is computed by an application that uses input values retrieved from several MIB counters.

In this chapter we present an aggregation of indicators model for abstracting the behavior of a network. This model is applied to the design of a management application that evaluates the state of an Ethernet network. Complexities that are similar to those of network management occur in other large systems, like market economies. The behavior of a stock market, for instance, is characterized by index functions like the *Dow Jones* average. Such indexing typically uses linear aggregation of a large number of variables, each providing a different microscopic measure of the system's state. Notice that such aggregation uses statically defined algorithms for its index functions, i.e., the definition of each index is stable for long periods of time. We apply a dynamic aggregation model to characterize the behavior of networks, using M_bD. Computed indexes reduce a large number of observed operational variables to a few simpler indicators of the system state that indicate important trends. The definition of each index function is dynamically delegated, and hence may change as frequently as network conditions require it.

Observation and Analysis of Networks

Many network management functions require real-time observation and analysis of the behavior of network elements. For instance, error rates at the communication links are computed in order to discover fault and performance problems. The network behavior is defined by thousands or even millons of variables that change rapidly. Therefore, the observation and analysis of the combined behavior of a network is a very difficult task.

Management applications need to use the observed information collected to resolve network problems. Such applications would need to make effective decisions based on vast amounts of real-time operational data. These decision processes are operators that compress network observation data input into a simpler decision output.

In current management frameworks, however, most of the data compression needed to evaluate management decisions is accomplished through manual processes. That is, network operators visually analyze the raw MIB data presented by an MIB browser to determine its meaning. Given the explosion of standard and vendor-specific private MIBs, and the increasing number of managed entities in each network, this manual task is extremely difficult. Developing effective technologies to compress management data at its source is, therefore, a central problem of network management.

Example: A Network Storm

Many network management problems are caused by complex interactions between the many components of distributed systems. For instance, consider a database server which maintains multiple TCP connections with remote clients. Assume that some of these connections pass through a particular T1 link. Suppose that a long burst of noise disrupts this link distorting several T1 slots. Frames traversing the link will incur many bit errors and will be lost. Link layer protocols will then invoke automatic frame retransmissions. If the noise is sustained, the retransmissions will overload the interface cards of the different hosts. This will eventually cause timeouts at the higher network layer, leading to TCP connection resets, and yet more retransmissions. Eventually, such a situation may cause a network *"storm"* scenario, i.e., the rapid escalation of cascading failures.

This example illustrates some of the difficult problems and complex interactions involved in managing distributed systems. Complex faults, as in the above example, often escalate in space and time. To handle such faults, management applications need to correlate observations from different network layers. For instance, they need to correlate the behavior of the TCP entities at the transport layer with the error rates at the link layer. For instance, if the noise is repeated or sustained, a management application may quickly modify the routing tables of intermediate nodes to avoid using the faulty link. Such applications need to be spatially distributed to enable real-time reaction to such problems. Some applications also need a central-

ized management process that concentrates on correlating the global observations performed by the distributed processes.

Observation Problems

The symptoms of complex network faults as in the above example, are often difficult to observe and to correlate. For instance, sometimes the faulty link is part of a leased network connection which is managed by another organization. It is usually not practical to continuously poll across network domains to retrieve potentially useful data. Furthermore, MIBs provide only partial approximations and sometimes even erroneous indications of the actual state of a device. The observed behaviors at the MIB are separated by the network from the locus of the applications computations. Network polling for MIB data results in delay, loss, and replication of values. SNMP applications must use observations that are predefined and hard-coded as part of the SNMP-agent at the device. Using remote delegation, network polling errors and perturbations can be significantly reduced and sometimes even completely eliminated.

Management Decisions

Network faults are often determined through *threshold decisions*. For instance, when the error rate in input packets exceeds a certain threshold, this may indicate a significant problem in the underlying medium. A management decision can disconnect a port on an Ethernet hub when its error traffic load exceeds a given threshold. Often, a number of indicators must be simultaneously considered in deciding how to handle a symptom. For instance, on a shared medium like Ethernet, excessive error rates may result from collisions due to a normal increase in traffic load. If only the error rate observation is considered, an application may overreact to the problem. A myopic decision of disconnecting the port may cause an even worse problem.

A Formal Model for Behaviors and Observations

Developers of management applications need a formal model and notation to properly describe and understand the behaviors of managed entities. Such a model should elucidate the roles of the different entities and computational processes involved in management computations. For instance, we need to differentiate between (1) the actual behavior of a device which is experiencing a problem, (2) the symptoms of the problem as seen by the observations computed by an MIB, (3) the remote observations of MIB variables performed via a management protocol, and (4) the computations performed by applications at the platform host.

This chapter introduces a formal model and its corresponding notation to describe the *sample behaviors* of *managed entities* and their observations by management applications. This model elucidates many of the shortcomings of current network management paradigms. In particular, we examine the observations of managed entities that are computed by SNMP MIBs, and characterize their problems and

limitations. The static approach of standard network management platforms to collecting data of managed entities wastes computing resources in observations that are not used by any application.

Index Functions for Compressing Management Data

A *health* function is an index function that combines real-time management data from composite observation operators to reach management decisions. Management applications may use these operators to make real-time decisions, e.g., to diagnose and correct element failures. We describe the design, implementation, and demonstration of a management application that uses such operators to provide a *health* index for a network. Observation operators must be adapted as (1) more knowledge about the network behavior is acquired, (2) the network resources and its usage policies evolve, and (3) as the expertise of the network human operators improves. Diagnostic procedures and corrective actions may be dynamically delegated as a result of an event. For instance, if the level of an Ethernet utilization becomes too high, the application may identify the device which is the source of most traffic and temporarily disconnect it from the network.

Current network management paradigms do not support the temporal distribution and spatial decentralization required to compute real-time health functions effectively. Health functions cannot be predefined as part of a static MIB, as they may vary from site to site and over time. Nor can they be usefully computed at centralized management platforms, since this can result in excessive polling rates, lead to errors due to perturbations introduced by polling, and miss the very goal of compressing data maximally at its source. Therefore, we use the M_bD approach to implement the health observation operators.

Chapter Organization

Section 4.2 introduces a formal notation to describe the behaviors of managed entities, and their observations by management software.

Section 4.3 describes some of the problems of computing observation operators using SNMP.

Section 4.4 describes management decision and correlation processes.

Section 4.5 describes index functions for compressing real-time management data.

Section 4.6 describes the design and implementation of an M_bD application that monitors and controls a LAN.

Section 4.7 summarizes the chapter and presents some conclusions.

4.2 Behaviors of Managed Entities

Management applications require *observation* data to understand the actual *behavior* of the managed entities. This section introduces a formal notation to specify the behaviors of managed entities (e.g., devices) and their observations by management applications. The use of a formal notation elucidates some of the distinctions between different types of management computations. We use this notation to define management operators that monitor and control managed entities.

4.2.1 Managed Entities

Let α denote an arbitrary managed entity of a networked device. For instance, α may be the Ethernet interface card of a given workstation. The *current state* of α is defined by a finite set of externally observable variable attributes, i.e., $State(\alpha) = \{\alpha_1, \alpha_2, ..., \alpha_n\}$. For example, α_1 may denote the length in octets of the last frame received at the interface. A variable may also represent a control routine of the managed entity which can be externally invoked by some interaction. For instance, α_2 may represent an error reset routine that can be invoked by the networked device. Such routines are typically invoked via a private control protocol between the networked device and the interface card over the internal bus.

4.2.2 Sample Behavior

Let $x = \alpha_j$ denote an arbitrary managed entity attribute of α. A *sample behavior* of x is a sequence $X = \{[t_x(n), x(n)]\}_n$. Here $t_x(n)$ denotes the time at which the n-th change in the state of x occurs. The state of x may change through an event such as a sample computation or communication of α that involves the variable x. $x(n)$ denotes the value of x after the n-th change.

For example, x_ι may denote the length in octets of the last IP frame received at a given interface (ι), and n counts frame arrival events. Suppose that a sequence of IP frames arrives at ι with the following lengths:

$$x_\iota(n) = \{345, 567, 779, 678, 333, 456, 322, 234, 378...\}.$$

Figure 4.1 shows the plot of the sample behavior X_ι. X_ι is a sample behavior of the arrival of these IP frames to ι, e.g.,

$$X_\iota = \{[1, 345], [3, 567], [8, 779], [13, 678], [17, 333], [21, 456], [23, 322], [30, 234], [34, 378], ...\}$$

Note that in this example, the attribute x is a scalar value, but in general, x may be a vector of entity attributes.

The notation $X(n) = [t_x(n), x(n)]$ denotes the *instantaneous value* of the sample behavior X associated with the n-th change. In the above example, $X_\iota(n)$ is a pair that includes the timestamp of the $n-th$ IP frame and the number of data octets in that frame. For instance, $X_\iota(2) = [3, 567]$. To associate behavior values with a

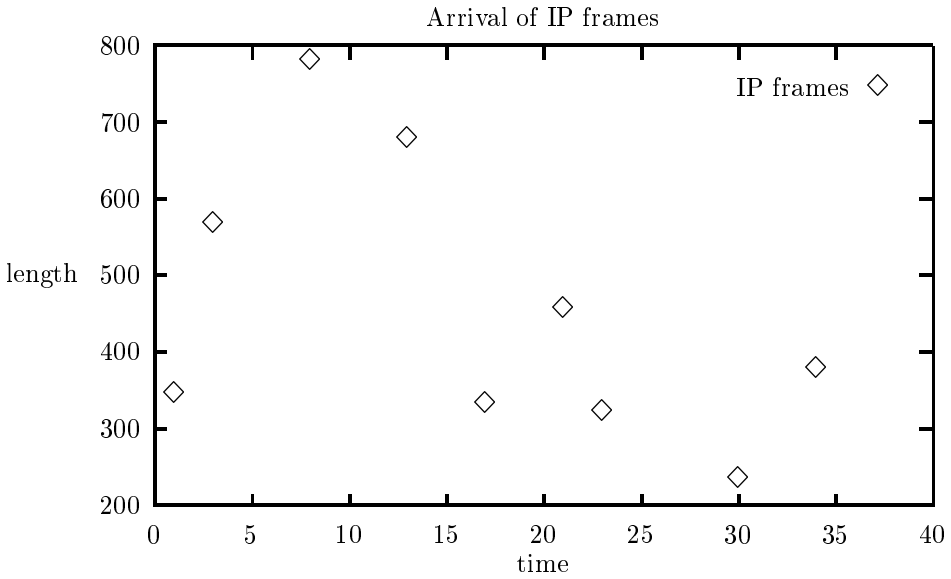

Figure 4.1: A Sample Behavior of IP frames

given time τ, we define $n_x(\tau) = max\{n|t_x(n) \leq \tau\}$. $n_x(\tau)$ is the ordinal number of the most recent event occurrence prior to time τ. For instance, $n_{x_\iota}(10)$ gives the number of IP frames that arrived before $\tau = 10$, which in the above example is 3.

4.2.3 Observations of Sample Behaviors

Observations of sample behaviors are defined by sequence operators that compute a function over the behavior's history. For example, the total number of data octets arriving at an interface ι via IP integrates the sample behavior over X_ι up to a given point in time. The initial history of a sample behavior X up to time τ is

$$X^\tau = \{[t_x(1), x(1)], [t_x(2), x(2)], ..., [t_x(n_x(\tau)), x(n_x(\tau))]\}.$$

An *observation* of a managed entity variable x is a computable functional $\mathcal{F}[X^t, t] = y$, where y is the value observed. An observation operator computes a sample behavior. For example, given the above sample behavior, the following is an observation of it by an arbitrary operator $\mathcal{Z}$:

$$\mathcal{Z}(X_\iota) = \{[1, 345], [8, 779], [13, 678], [21, 456], [34, 234], ...\}$$

An *observation process* is given by a mapping $\mathcal{F} : \{[t_x(n), x(n)]\} \mapsto \{[t_y(n), y(n)]\}$ where $y(n) = \mathcal{F}[X^{t_y(n)}, t_y(n)]$. Let $\mathcal{Y}$ be an arbitrary observation operator. If the time of the k-th observation of $\mathcal{Y}$ is $t_y(k)$ and the value observed is $y(k)$, then the sequence $\{[t_y(1), y(1)], [t_y(2), y(2)], ..., [t_y(n), y(n)]\}$ represents the sample behavior of the observations. Figure 4.2 displays the original sample behavior and an observation process sample behavior. Notice that there are several IP frames that are not

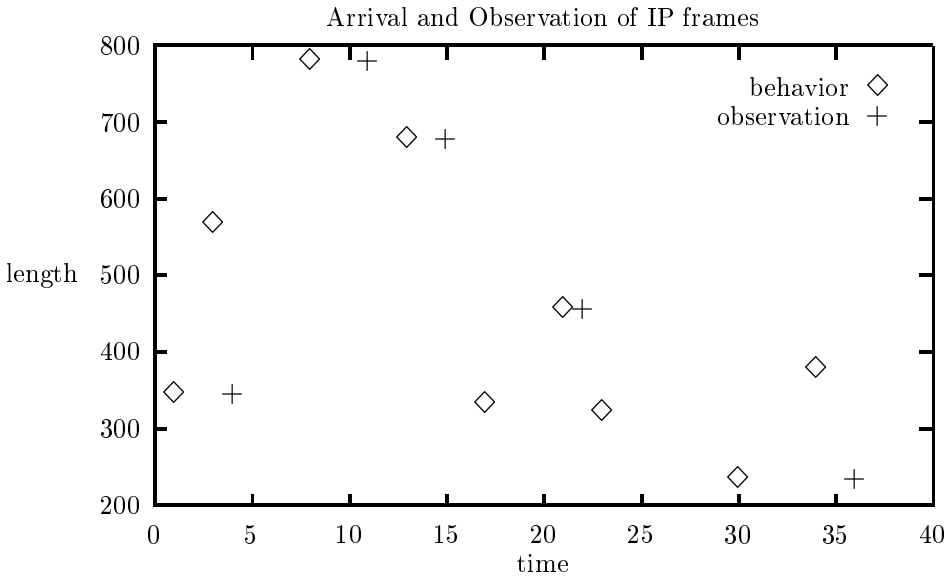

Figure 4.2: A Sample Observation of IP frames

observed, i.e., the observation misses them. The following sequence is computed by an observation operator $\mathcal{Y}$, over the sample behavior X_ι defined above.

$$\mathcal{Y}\mathcal{Z}(X_\iota) = \{[4,[1,345]],[11,[8,779]],[15,[13,678]],[22,[21,456]],[36,[34,234]],...\}$$

Sample behaviors are often observed via observation operators that are cumulative counters. Sample observations over counters lose less information, (e.g., the timing of the changes), since the counters aggregate the data. The MIB-II variable `ipInUnknownProtos`, for instance, counts (modulo 2^{32}) *"the number of locally-addressed datagrams received successfully but discarded because of an unknown or unsupported protocol"* [McCloghrie and Rose, 1991]. A counter of X may be defined as $\mathcal{C}(X^t) = \sum_{j \le n_{x(t)}} x(j)$. Thus, if $\mathcal{U}(x(n))$ is defined as 1 when the protocol is unsupported or unknown, and 0 otherwise, then `ipInUnknownProtos` $\approx \sum \mathcal{U}(x(n))$.

Operators can be Defined Recursively

$Y = \mathcal{C}X$ can also be defined recursively via

$$t_y(n) = t_x(n), \quad y(n+1) = y(n) + x(n+1).$$

The first part means that the counter is updated (observed) at times when x changes, and the second part defines the counter values recursively. In a similar manner, derivatives of behaviors may be defined recursively. For example, $Y = \mathcal{E}^k X$ is a moving average observation over a window (number of events) of length k, and is defined by

$$t_y(n) = t_x(n), \quad y(n) = \sum_{n-k<j\le n} x(j)/k.$$

4.2.4 Generic Observation Operators

Many observations depend only on the temporal aspects of a behavior X and are, therefore, generic. Let us consider a few examples (in all cases $t_y(n) = t_x(n)$):

- $Y = \mathcal{N}X$ is defined by $y(n) = n$ and measures the total number of events. This observation is approximated in SNMP MIBs by counters.
- $Y = \mathcal{A}X$ is defined by $y(n) = t_x(n) - t_x(n-1)$ and measures the interarrival time for $x(n)$. For example, $\mathcal{A}$(`ifInErrors`) measures the time between errors in interface input packets.
- $Y = \mathcal{R}X$ is defined by $y(n) = 1/[t_x(n) - t_x(n-1)]$ and measures the rate of events arrivals. For example, $\mathcal{R}$ (`ifInErrors`) measures the input error rate of a given interface.

Observation operators may also be composed to produce new observations. For example, $\mathcal{E}^k\mathcal{A}X$ provides the average interarrival time of X over a window of length k.

4.3 MIB Observations

4.3.1 Deriving MIBs as Observations of Behaviors

Each MIB defines the data that it makes accessible to applications via a management protocol. Consider a real managed entity α with a sample behavior X_α. An SNMP-agent applies an observation operator $Y = \mathcal{M}_\alpha$ to produce the value of an MIB variable $\mathcal{M}_\alpha X_\alpha$. For example, consider a router device ρ, with an attribute variable $x_{\rho,i}$ that denotes the number of octets in the output queue of the i interface card. The SNMP MIB-II [McCloghrie and Rose, 1991] variable `ifOutQLen` is computed by an observation operator that observes the length of such an output packet queue.

An MIB is, therefore, a set of managed entities observations $\{\mu_1, \mu_2, \mu_3, ...\}$, where each μ_i is defined by an observation $\mathcal{M}_i X_i$. Some of the μ_i are defined by observations over attributes that are constant or change very infrequently. For instance, the MIB-II variable `ipForwarding` indicates if the device is acting as a gateway or not. Such an attribute is unlikely to change frequently. Most interesting attributes like `ifInOctets` do change rapidly.

Example: Sampling a Counter

In SNMP, observations of operational variables are accomplished via counters and gauges. A counter represents a cumulative (integral) of an operational variable. Typically, however, only the change in the counter value provides a useful indication of the network state. For example, the MIB-II counter variable `ifInOctets`

(`ifOutOctets`) counts modulo 2^{32} the total number of bytes received (sent) by an interface since the initialization of the device. The rate at which these counters change is a useful indication of load on a network segment. Notice that counters may wrap around 2^{32}. For instance, `ifInOctets` can wrap around in a minimum of 3.5 seconds for a SONET OC-192 interface at 10 Gigabits per second. Therefore, observation operators may need to perform very frequent samplings of the sample behaviors of some managed entities. Obviously, it will be very taxing for any host to sample the SONET frames at such speeds. Therefore, MIB observations will only provide approximations to the real sample behavior of such managed entities.

4.3.2 Observation Operators Introduce Delays

The delays associated with the MIB observation operator $\mathcal{M}$ will vary depending on several factors. If the MIB is located at the same host where the managed entity resides, this delay can be relatively low. For instance, an SNMP agent may implement $\mathcal{M}$ via direct retrieval of operating system kernel data structures, by accessing shared registers or shared memory. However, if the managed entity changes frequently, the local host will not be able to evaluate $\mathcal{M}$ at a high enough sampling rate to capture the complete sample behavior.

Observation Sampling

Consider $X^\omega = \{[t_1, x_1], [t_2, x_2], ..., [t_n, x_n]\}$, a finite real sample behavior at time ω. $\mathcal{M}$ will produce an approximation, $\mathcal{M}X^\omega = \{[t_{i_1}, x_{i_i}], [t_{i_2}, x_{i_2}], ..., [t_{i_k}, x_{i_k}]\}$. The number of elements (n) in the sequence X^ω is significantly larger than that of $\mathcal{M}X^\omega$, (i_k). That is $i_k = |\mathcal{M}X^\omega| < |X^\omega| = n$. If $\mathcal{M}$ is applied too infrequently, some of the changes in the sample behavior X may be lost, and therefore the sample behavior of the entity α may only be partially observed in the MIB. The quality of such approximations depends on MIB implementation decisions. For instance, an SNMP agent may invoke the $\mathcal{M}$ operator at a predefined frequency for some variables and only upon receiving an external request for others.

Proxy Agent Example

As another example, an MIB can be implemented by a *"proxy agent"* which observes managed entities at remote devices. In this case, $\mathcal{M}$ is itself bound by the behavior of the network resources needed to observe a remote managed entity. Proxy SNMP-agents, in particular, may need to poll remote devices over slow communication links to observe managed entities. For example, an SNMP-agent may be a proxy for a remote modem that does not support SNMP. The polling delay of $\mathcal{M}$ may be significant when compared with the required frequency of observation sampling for a real-time diagnostics application. The polling delay behaves as a random variable due to the behavior of network resources, e.g., phone link errors. Therefore, proxy sampling can introduce a substantial error in the observation of $\mathcal{M}$.

4.3.3 Management Applications Compute Observations

Consider generic observation operators like interarrival time $\mathcal{A}$ or event rate $\mathcal{R}$, which were defined in Section 4.2.4. For instance, $\mathcal{R}$ is useful to compute the error rate of frames arriving at a given interface, e.g., $\mathcal{R}$(`ifInErrors`). Such generic temporal observation operators are not computed in SNMP's MIB-II. The SNMP approach is that such observation operators should be derived by management applications from other MIB variables by polling observations and arithmetic computations.

Only by polling the MIB-II variable `sysUpTime` can a manager obtain an approximate clue regarding event arrival times. `sysUpTime` is an observation operator that can be used as a substitute timestamp for sample behaviors observations[1]. For example, at any moment, the value of the pair [`sysUpTime`, `ifOutQLen`], represents the last instantaneous value of the sample behavior defined by the length of the interface queue.

Temporal observations may only be approximated by SNMP polling. Management protocols like SNMP support a polling operator $\mathcal{P}$ to bring observed data from the MIBs to their platforms. Therefore, MIB data access by NOC management applications can be described as the composition of the respective operators, $\{\mathcal{P}\mathcal{M}_\alpha X_\alpha\}$.

Management Applications Compose Observations

Computations by management applications often apply additional operators to these polled values, say $\{\mathcal{U}\mathcal{P}\mathcal{M}_\alpha X_\alpha\}$. An application may compute long term statistics of MIB variables and use them to compare the current behavior of the network. For instance, a management application may collect a daily sample behavior observation of the load on a network segment, by computing $\mathcal{U}_1\mathcal{P}$(`ifInOctets` + `ifOutOctets`). $\mathcal{U}_1$ may perform a standard deviation computation that is used to analyze periodic trends. For example, a LAN segment may become daily loaded at 3AM due to file system backups. An application may define another operator, $\mathcal{U}_2$ that compares the current sample behavior with an historic benchmark, to discover anomalous behaviors. Thus, $\mathcal{U}_2$ will realize that the 3AM peak utilization does not require the generation of an alert message.

The observed behaviors at the MIB are separated by the network from the locus of the applications computations. In other words, $\{\mathcal{M}_\alpha X_\alpha\}$ is typically computed at close proximity to the managed entity α, while the application computations $\mathcal{U}$ are typically computed at centralized NOC hosts. In SNMP, a polling operator $\mathcal{P}$ is used to bridge this spatial separation. Since polling introduces observation perturbations, the fidelity of the $\mathcal{U}$ computations may suffer. For instance, in the above example, the benchmark statistical computations may be polluted by polling errors or delays.

[1] `sysUpTime` measures the time in hundredths of a second since the network management portion of the system was last re-initialized.

Polling Observations Introduce Perturbations

Let us analyze the reasons for polling perturbations. Let $Y = \mathcal{P}X$ denote the observations produced by the polling operation over X. Ideally, the polling operator $\mathcal{P}$ should provide a perfect approximation to the identity observation operator $\mathcal{I}$, i.e., $\mathcal{I}X = X$. However, $\mathcal{P}$ is seldom identical to $\mathcal{I}$. Only for observations of sample behaviors that do not change. For instance, an MIB variable like `sysDescr` represents a fixed attribute, the description of the device, that does not change unless the MIB is reconfigured. $\mathcal{P}$ will differ from $\mathcal{I}$ for most "interesting" observations, i.e., those which have non-trivial sample behaviors.

Delay, Loss, and Replication

Polling results in three kinds of perturbations of the sample behavior being observed: delay, loss, and replication of values. Observations have delays which may depend on many independent factors, such as network traffic, the load on the server's host, and so on. Some of the original values may be lost when polling is not performed frequently enough to track changes in x. Polled values may also be lost due to network errors. Indeed, some management protocols (e.g., SNMP) use non-reliable transport protocols (e.g., UDP) to convey management data. Furthermore, when polling times are too frequent, the same values of x may be polled, resulting in duplication. Such duplicate values may skew the computations of statistics over the sample behaviors. As a result of these perturbations, a polling operator $\mathcal{P}$ may retrieve a sample behavior sequence that substantially differs from the ideal observation operator $\mathcal{I}$.

4.3.4 Static MIBs Waste Resources

MIB implementations define the collection of observations $\mathcal{M}$ a priori and independently of their use. We call this a *static approach* to the collection of device operational data. Since the need for operational data cannot always be predicted, many observations are collected and stored for potential access by management applications. This results in the collection of large amounts of data that may never be used.

In most MIBs, applications can not control the execution of the observations $\mathcal{M}$. For instance, an MIB at a device may execute the observation of its interfaces at predefined intervals of time, say every 10 seconds. Thus, the exact meaning of the operator $\{\mathcal{M}_\alpha X_\alpha\}$ depends on the MIB implementor's design decisions. Some device agents may allow a startup setting of the frequency of observations. Still, they do not allow remote management applications to dynamically modify this frequency to suit their needs.

One may ask, how good is the static observation paradigm in supporting the computations of network decisions? There are two major problems of concern regarding this paradigm: (1) the rigid definition of MIB objects and (2) the use of polling to access management data.

MIB Objects are Rigidly Predefined

Observational resources are statically allocated to collect information that may never be used. Relevant, useful new observations cannot be dynamically added. Thus, applications must compute approximate observations, which may have significant errors due to MIB implementation decisions. For example, MIB-II variables capture error observations via counters, e.g., `ifInErrors`. Let X denote the behavior captured by an error counter. The error arrival rate is given by the observation $Y = \mathcal{R}X$, where

$$t_y(n) = t_x(n), \quad y(n) = [x(n) - x(n-1)]/[t_x(n) - t_x(n-1)].$$

Since the MIB only records the values of $x(n)$, the values of $t_x(n)$ are not directly observable. Note that it is possible to approximate $t_x(n)$ by polling `sysUpTime`.

For example, the interface input error rate can be approximated as:

$$\mathrm{E} = \Delta\texttt{ifInErrors} \ / \ \Delta\texttt{sysUpTime}.$$

However, `sysUpTime` may differ from the time at which the errors occurred, and when the MIB variables were last updated. This is likely to happen if the agent is acting as a proxy. Thus, error rates and similar observations cannot be computed directly, but must be approximated.

Polling Perturbs Management Computations

A second problem is that of computing linear decisions based on complex observations obtained via polling. Even if all the behavior variables accessed by a management function were available in an MIB, it is still necessary to apply polling over the network to retrieve them to a manager which can compute $\mathcal{H}$. Since, in general, $\mathcal{PH} \neq \mathcal{H}$, the perturbation introduced by polling corrupts the values available to compute $\mathcal{H}$. Limitations on the maximum rate of polling and the randomness in polling delays restrict the computations of management decisions. Moreover, they may result in management applications taking wrong decisions, or not taking action when required.

For example, security management applications use MIB variables to monitor transport connections. To track which remote systems access resources via TCP, for instance, `tcpConnTable` can be used [Leinwand and Fang, 1993]. An intruder, however, may need only a brief connection to gather information. If polling does not occur within this period, the record of the intrusion may be lost, and management actions will not be taken.

4.3.5 M_bD Provides Control of Predefined Observations

M_bD addresses the limitations of static observations by providing dynamic control of observations. An M_bD-server supports the dynamic control of predefined observations, and their dynamic composition and configuration. In contrast, SNMP

applications must use observations that are predefined and hard-coded as part of the SNMP-agent at the device. Using delegation, network polling errors and perturbations can be significantly reduced and sometimes even completely eliminated. Observation processes may be invoked and terminated via RDS thread control operations. This enables M_bD applications to control the allocation of device resources to observation tasks. Applications can flexibly monitor device behaviors according to their interests, without wasting computing cycles and network bandwidth on unnecessary observations. New observation operators can be dynamically defined and delegated as requirements evolve.

4.4 Threshold Decisions and Observation Correlations

Management applications use *decision processes* to trigger actions in order to handle management problems. Network management decisions may be described as Boolean functions defined over the universe of management observations. Such functions may need to compress large amounts of real-time observations. For instance, a fault management application reacts to a device fault by performing some corrective action, e.g., rebooting the device. To reach the correct decision on real-time, the application needs to compress large sample observations, filtering out irrelevant samplings, and extracting the problem indicators which are relevant to a particular fault.

4.4.1 Faults can be Detected by Threshold Decisions

Network faults are often detected through *threshold decisions*. A threshold decision is defined by a Boolean function of management observations,

$$\mathcal{D} : \{\mathcal{U}\mathcal{P}\mathcal{M}_\alpha\mathcal{X}_\alpha\} \mapsto \{\text{Yes}, \text{No}\}.$$

For instance, when the error rate in input packets exceeds a certain threshold, this may indicate a significant problem in the underlying medium. A management application may decide to disconnect a given port on an Ethernet hub when its error traffic load exceeds a given threshold. In this case, α is the Ethernet port, $\mathcal{X}_\alpha$ is the actual sample behavior of errors on α, $\mathcal{M}_\alpha$ is the observation of this sample behavior as computed by the corresponding MIB variable at the hub, $\mathcal{P}$ is the observation operator implemented by the SNMP protocol, and $\mathcal{U}$ is the management application computation that evaluates error rates and takes action to correct the existing problem.

Inputs to Decision Processes

Often, a number of indicators must be simultaneously considered in deciding how to handle a symptom. For instance, on a shared medium like Ethernet, excessive

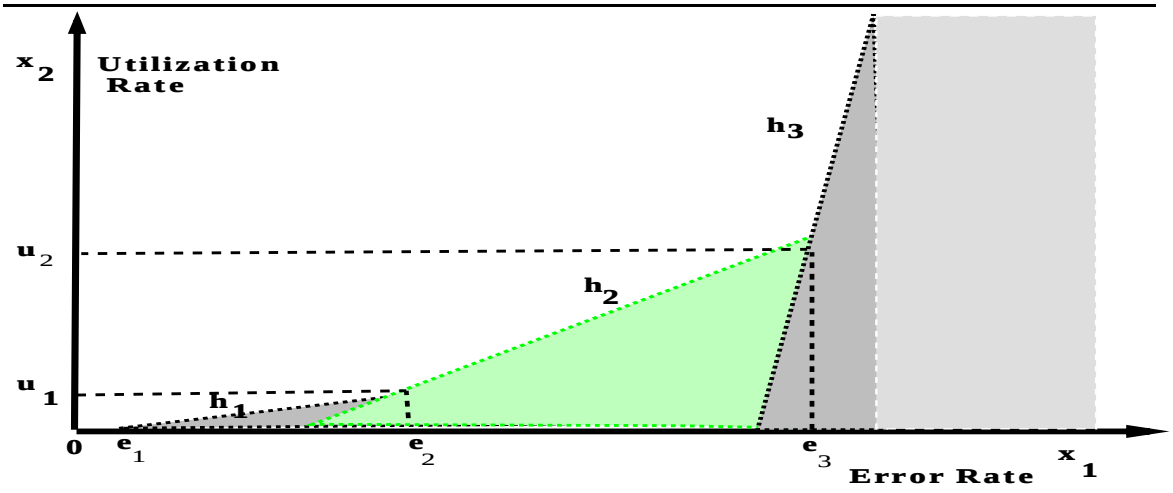

Figure 4.3: Utilization and Error Rates Domains

error rates may result from collisions due to a normal increase in traffic load. If only the error rate observation is considered, an application may overreact to the problem. A myopic decision of disconnecting the port may cause an even worse problem. To avoid such problems all the relevant indicators should be used as inputs to the decision process. In this case both the error and utilization rates must be input.

Linear Decisions are Fast to Compute

A linear weighted measure is an efficient method to combine managed variables in a threshold decision. For example, consider $\vec{x} = (x_1, x_2, ..., x_k)$ as a collection of managed variables involved in a decision. A simple linear index function h is defined by the scalar product $h(\vec{x}) = \vec{w} \cdot \vec{x} = \sum_{1 \leq i \leq k} w_i x_i$, where $\vec{w} = (w_1, w_2, ..., w_k)$ defines a set of weights corresponding to each observation. Such an index function is an aggregate measure of network behavior. Linear decisions are particularly attractive for providing fault indications which can be used as input to global index functions. A *linear threshold decision* is defined by $h^+ = \{\vec{x} | h(\vec{x}) = \vec{w} \cdot \vec{x} \geq 0\}$. Values of $\vec{x}$ in h^+ define exceptional events possibly indicating symptoms of problems[2].

Error and Utilization Rates Example

An index function h (possibly vectorial-valued) defines an observation over behaviors X as $Y = \mathcal{H}X$ where $t_y(n) = t_x(n)$, and

$$y(n) = \begin{cases} 1 & \text{if } h(x(n)) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

[2]Decisions of the type $h(x) = \vec{w} \cdot \vec{x} \geq h'$ may be easily reduced to the case above by considering $h' = \vec{w}' \cdot \vec{x}'$ where $\vec{w}' = ((w_1, w_2, ..., w_k, h')$ and $\vec{x}' = (x_1, x_2, ..., x_k, -1)$.

The operator $\mathcal{H}$ results in a value 1, whenever the values of x fall within the unhealthy domain h^+. When the observed behavior X wanders into the unhealthy region, it may be necessary to invoke appropriate management procedures to handle potential problems.

Let x_1 denote the input error rate and x_2 the utilization rate associated with a given interface. An index function that correlates utilization and error rates may be defined as follows.

$$H = \begin{cases} 0 & x_1 < e_1 \\ h_1(x_1, x_2) & e_1 \leq x_1 < e_2 \\ h_2(x_1, x_2) & e_2 \leq x_1 < e_3 \\ h_3(x_1, x_2) & e_3 \leq x_1 \end{cases}$$

Error rates lower than e_1 are considered insignificant. An error rate in the ranges defined by e_1 and e_2, for example, is considered significant if the utilization rate is lower than $h_1(x_1, x_2)$. The domain where faults are indicated is bounded by 3 linear functions as depicted in figure 4.3. The fault indication region is defined by the intersection of the half-planes decided by each of these index functions. In other words, if $h \equiv (h_1, h_2, h_3)$, the fault indication domain is $h^+ = \{\vec{x} | h(\vec{x}) \geq \vec{0}\}$.

Sustained and Intermittent Problems

Sometimes management actions need to be invoked only when unhealthy behavior is sustained for a period of time, or when it is repeated intermittently. For example, a device may contain operational control mechanisms that provide temporary relief from sustained problems. For instance, an ethernet interface card will perform an exponential back-off when confronted with collisions. Thus, intermittent problem indications may arise.

To avoid spurious alerts, a threshold excess must be sustained over a sufficiently long time window. For example, a hysteresis mechanism should be implemented to limit the generation of alarms. If the observed unhealthy behavior fluctuates, an alert should not be generated. These temporal problem indicators may be captured by appropriate observation operators applied to the output of the $\mathcal{H}$ observation. Sustained problems, such as unhealthy behavior for a period of duration δ, may be detected by $Z = \mathcal{P}Y$ where

$$z(n) = \begin{cases} 1 & y(j) = 1 \text{ for } n_y(t_y(n) - \delta) \leq j \leq n \\ 0 & \text{otherwise} \end{cases}$$

Applying then $Z = \mathcal{P}\mathcal{H}X$ will provide observations of sustained unhealthy behaviors.

4.4.2 How to Classify Weights to Define Index Functions

Index functions depend on local administrative policies, and must be adapted to each installation site. Fault indications, for instance, may vary for each network type (e.g., token-ring or ethernet), installation (e.g., educational or military), and

time period (e.g., morning or night). Index functions need to be properly established to reflect these differences. To define an index function requires computing the linear weights $\vec{w}$ used in threshold decisions. This definition can be done manually by the network administrators, and it could be assisted by the use of some software. Research work in artificial intelligence has dealt with this problem, e.g., *pattern classification* and *perceptron training*. The following paragraphs briefly outline some of their main characteristics.

Pattern Classification

Pattern classification programs often combine several features of a given stimulus in order to determine its category [Rich and Knight, 1991]. It is often difficult to know a priori how much weight should be attached to each feature. One way of finding appropriate weights is to begin by using estimates, and let the program modify the settings. Good (poor) predictors should have their weights increased (decreased) until correct classifications are achieved. Similar learning techniques have been used for game playing programs, e.g., Samuel's checkers [Cohen and Feigenbaum, 1981]. There are several issues that must be addressed for this type of technique. Among them, (1) when and by how much to adjust the values of $\vec{w}$, and (2) when to add, delete, or replace any of the basic terms of the function, e.g., a x_i.

Perceptron Training

The problem of computing $\vec{w}$, can be considered an instance of the problem of perceptron training [Duda and Hart, 1973]. A number of known algorithms may be used to train such a linear function. The Least Mean Square (LMS) algorithm, for example, adapts the weights after every trial, based on the difference between the actual and desired output [Cohen and Feigenbaum, 1981]. The single layer perceptron model is appropriate when computing linearly separable regions. Multi-layer perceptrons are needed for complex cases, e.g., when decision regions can not be separated by a hyperplane [DARPA, 1988]. In a more general scenario, multiple index functions may be simultaneously employed, each providing indications of different possible problems. Threshold decisions for these functions can be combined using multi-layer perceptrons.

4.4.3 Correlations Between Observations

Example: Storm of Noise Leads to Retransmissions

Let us consider a network storm fault scenario, i.e., the rapid escalation of cascading failures. Consider a database server which maintains TCP connections with remote clients over a T1 link. Multiple TCP connections are multiplexed into the T1 slots. Suppose that a long burst of noise disrupts the link garbling T1 slots. Frames traversing the link will incur bit errors with high probability and will be

lost. Logical link level protocols above the T1 layer will then invoke automatic frame retransmissions.

The burst of signal-level link noise is translated by higher layer protocols into a burst of retransmission tasks. The interface processor serving retransmission tasks will have its queue overloaded. This can potentially lead to the processor thrashing. Higher layer transport entities will time out on delays in the responses. These timeouts will cause a burst of corrective activities (e.g., reset connections). Again, a lower layer burst of exceptional processing is translated into a higher layer burst. In summary, protocol stacks tend to propagate problems up to entities at the higher layers and through the network. This propagation can lead to rapid escalation of faults.

Suppose that an application observes the bit error rates at the physical level (`ifInErrors`), the size of the interface queue at the link level (`ifOutQLen`) and the rate of TCP connections reset (`tcpEstabResets`) at the transport level. These observation processes are depicted in figure 4.4. The behavior of the link-level queue is correlated with error rates occurring sometime earlier. Similarly, the rate of connection resets is correlated with the length of the link-level output queue.

Correlations are not Observed in MIBs

The storm escalates through the formation of temporal-spatial correlations among behaviors at different layers and locations. However, as noted in previous sections, the real sample behavior of such processes is typically not observed. Instead, snapshots representing integrals of changes (accumulated through counters) are collected in MIBs. For instance, in figure 4.4, the error count collected in the MIB represents the area under the error rate curve.

To properly diagnose the condition of the network such correlations should be detected. To observe these correlations, a management application must first compute the appropriate sample observations (error rates, transmission resets), and then evaluate the correlations between them. For example, the correlation between two sample observations may be measured as a function of their covariance.

Thresholds are Used to Identify Faults

Threshold detection processes may sometimes hide the relationship between correlated events. For example, a management process may detect a threshold for the effect (TCP resets) but not for the cause (frame errors). Events may also be reordered in time. For instance, a management application may detect an effect event via a threshold function that is computed over a longer interval than that of the causal event. Threshold detection may cause correlated observations of unrelated events, e.g., via detection of spurious samples. Threshold events detection may complicate the correlation of observations, leading to incorrect diagnostics and inappropriate management actions.

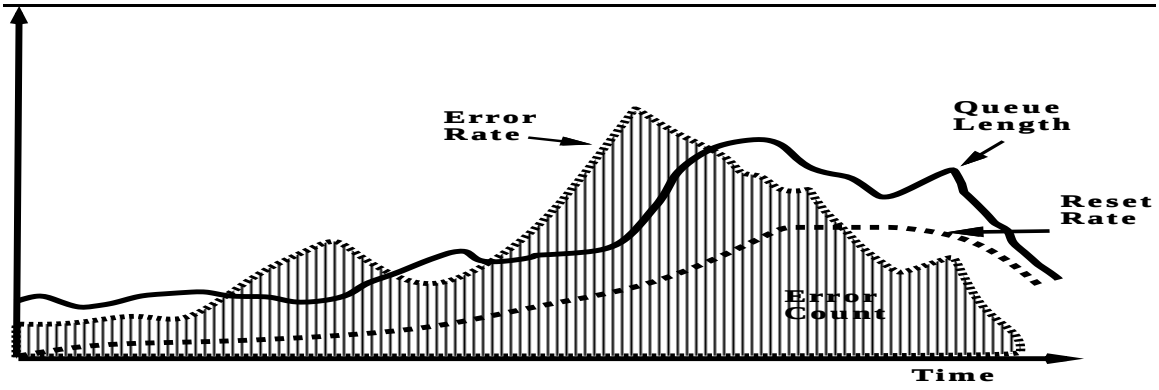

Figure 4.4: Temporal Behavior Correlations.

Example of Threshold Decorrelation Problem

Assume that for a given network router, queue length excesses are detected with greater sensitivity than link error rates. An alert on queue length may be generated without its corresponding error-rate alert. Such an alert may be analyzed under the incorrect assumption that error rates are normal. This may lead to the wrong conclusion regarding the source of the problem. In general, many alerts may be generated by a single fault, rendering the identification of the fire that caused the smoke very difficult. Sometimes, the smoke may result from the observation process itself, not from a real fire.

Faults Across Domains are Difficult to Detect

Networks may be operated by separate organizations, each responsible for a different domain. For example, a leased T3 link is managed by a telephone company, while the layers above it are operated by an academic institution. The network administrator of the academic institution may be unable to observe the sample behaviors of the link layer elements. Similarly, the telephone company operators may be unable to observe the behaviors of higher layer entities in the end user domain. Thus, faults escalating across domain boundaries are particularly difficult to detect. Section 5.3.3 presents a detailed example of management correlation across administrative domains.

MbD Helps the Evaluation of Correlations

Complex faults as those illustrated above are characterized by fast fluctuations in their observed symptoms. Such faults are difficult to observe and hence are difficult to correlate. Their symptoms provide only partial and sometimes even erroneous indications of the network state. Fault and performance management applications require an analysis of such observations to identify their causes. Section 4.3 showed

that observations and interpretation of behaviors are arduous to perform using a centralized paradigm of network management. Therefore, systematic fault and performance management are difficult to achieve and are mostly ineffective. Spatial and temporal distribution of the observation operators and their interpretation is needed to overcome these problems.

4.5 Index Functions for Compression

A distributed system needs to support effective management decisions based on vast amounts of real-time operational data. These decision processes are operators that compress sample behavior observations into a simpler decision. In current network management frameworks, much of the compression needed to evaluate management decisions is often accomplished through manual processes at the NOC. That is, network operators must visually analyze the raw MIB data presented by an MIB browser and determine its meaning. Given the explosion of standard and vendor-specific private MIBs and the increasing number of managed entities in each network, this manual task becomes extremely difficult. Developing effective technologies to compress management data at its source is, therefore, a central problem of network management.

One method of compressing operational data is to compute index functions, reducing a large number of observed operational variables to a single indicator of the system state. This is similar to the use of indexes to reflect the state of complex systems, such as the *Leading Economic Indicators* (LEI) index[3]. Such indexing typically uses a linear aggregation of a large number of variables, each providing a different microscopic measure of state. For instance, the LEI includes 11 indicators, such as "average weekly initial claims to unemployment insurance", "building permits for new private housing", and "index of 500 common stock prices".

4.5.1 MIB Computations Using SNMP

MIB variables can be combined to yield useful status indicators. For example, the utilization of an interface at time `t=sysUpTime` can be defined as

$$U(t) = \frac{(\texttt{ifInOctets} + \texttt{ifOutOctets}) * 8}{(\texttt{ifSpeed} * \texttt{sysUpTime} * 100)}$$

where `ifInOctets` (`ifOutOctets`) gives the total number of bytes received (sent), and `sysUpTime` is multiplied by 100 to yield units in seconds. This measure provides an average sense of utilization over a time window since the device was reinitialized. A useful indication of the instantaneous network state is provided by the derivative $u(t) = dU(t)/dt$. A derivative such as $u(t)$ may be approximated by frequent sampling of the respective managed variables and computing their changes in U. Index functions will typically utilize linear combination of such rates.

[3]See http://bos.business.uab.edu/forecast/lei.htm.

Similarly, one can establish measures of instantaneous error rates to capture additional micro-state indications. For example, the rate of input errors is $e(t) = dE(t)/d(t)$, where $E(t)$ is the percentage of input errors to packets delivered, and can be evaluated as

$$E(t) = \frac{\texttt{ifInErrors}}{(\texttt{ifInUcastPkts} + \texttt{ifInNUcastPkts})}.$$

Here `ifInErrors` counts the number of inbound packets that contained errors which prevented them from being delivered to a higher-layer protocol, and `ifInUcastPkts` (`ifInNUcastPkts`) count the number of subnetwork-unicast (nonunicast) packets delivered to a higher-layer protocol.

4.5.2 Index Health Functions

A *health* index function is a linear weighted function of management variables that aggregates micro-measures of local network state. Let us consider the state of a LAN which is defined by a concentrator hub with n interfaces. A simple health index function $\mathcal{H}$ may combine error and utilization rates. For example, $\mathcal{H}(\vec{e}, \vec{u}) = \vec{A}\vec{e} + \vec{B}\vec{u}$ where $\vec{e}$ and $\vec{u}$ represent error rates and utilization rates, and $\vec{A}$ and $\vec{B}$ are their corresponding weight vectors. This health function can provide a useful aggregate measure of the network state. In general,

$$\mathcal{H}(\vec{f_1}, \vec{f_2}, ..., \vec{f_k}) = \sum_{j=1,k} \vec{W_j}\vec{f_j},$$

where $\vec{f_j} = (f_j(1,t), f_j(2,t), ..., f_j(n,t))$ represents the evaluation of $f_j(t)$ at the n interfaces, and $\vec{W_j}$ is the corresponding weight vector. By compressing information at the hub, the load on the NOC hosts is significantly reduced. In contrast, platform-centric management computations require continuous polling of vast MIB information into the NOC computers and its manual interpretation.

4.5.3 A Health Function cannot be Statically Defined

Current network management paradigms do not support the flexibility and decentralization required to compute health functions effectively. Standard network management approaches require a priori knowledge of what algorithms are mapped into statically defined objects. The actual definition of what constitutes a *healthy* network can not be fixed for all networks. Such a definition depends on the specific configuration and administrative policies of each network installation. These parameters vary among different networks and during different times within the same network. For example, the rate of faults for a large WAN may be considered highly unusual for a small LAN. Utilization levels or delays which are normal for an academic department may be unacceptable for a hospital unit.

The Definition of Health evolves over Time

Even for a given network, the quantitative measure of health must vary dynamically, since over different time periods the normal metrics of the environment change. For example, the number of devices such as personal computers being used depends on the time of the day and the day of the week. Besides, the configuration of a network is constantly evolving as elements are added, replaced or eliminated. Therefore, health functions can not be statically defined, e.g., as part of an SNMP MIB. Similarly, a health function could not be usefully incorporated as part of an OSI managed object. While the OSI management framework permits encapsulation of functions within managed objects and their remote invocation by managing entities, these functions must be statically bound to a managed object at its design time.

Health Functions should be Distributed

A health function cannot be effectively computed by NOC hosts. The rates of polling required to aggregate the variables used may far exceed the platform's processing capability. In the example above, suppose that N devices (e.g. $N = 200$) are polled every s seconds (e.g. $s = 0.1$). The aggregated polling rate is then N/s, e.g., 2000 SNMP requests per second. As discussed earlier, polling through the network introduces random perturbations in approximating temporal derivatives of managed variables. This leads to observation errors and thereby potential hazards in making decisions. The goal of data compression is to reduce data volume at its source. Therefore, the observation operators that compose such functions must be spatially distributed to the devices.

The following section presents the design and implementation of an application that implements distributed health functions.

4.6 Health of a Distributed System

Developing appropriate metrics for computing an index function that represents the "health" of an heterogeneous network is not an easy task. In part, this is due to the semantic heterogeneity of the sample behaviors of network devices. In addition, the algorithms used to reduce sample observations into relevant information need to change frequently. Observation operators must be adapted as (1) more knowledge about the network behavior is acquired, (2) the network resources and its usage policies evolve, and (3) as the expertise of the network human operators improves. A health application needs both spatial and temporal distribution to adapt to evolving requirements. Dynamic application configurability is needed for simple parameters, such as thresholds, and also for the algorithmic logic that is embedded in the programs that implement the application.

The health application evaluates sample observations over numerous network elements and computes a high level abstraction of the operation of the network. Since

centralized management is unsuitable to compute health functions, their evaluation must be dynamically distributed. Delegating health functions to an M_bD-server enables the compression of observation data at its source. M_bD permits flexible changes in health functions to reflect specific behaviors at different sites and times. Delegated threads have direct access with minimal delay to the operational values of managed variables. Hence they accomplish greater precision in observing sample behaviors than would be possible using a centralized approach. M_bD reduces the rates of polling needed and restricts it mostly to times when problems are identified via aggregated index functions. Such functions are evaluated locally at the devices, and they continue to execute during critical stress times when NOC hosts have difficulties in reaching the devices. When an operational problem is detected, its delegated threads can invoke automatic correcting actions and produce event reports.

4.6.1 Components of the Health Application

The distributed "health" application consists of the following components:

- *Manager* application processes can dynamically reconfigure the distributed health application. They present evaluation reports on a graphical user interface so that operators can see a high level view of the network's operational state.
- A *Health* agent, HDMPI, receives reports from the observers and evaluates a higher level abstraction ($\mathcal{H}$) of the state of the network.
- *Observers* monitor the network elements, produce quantitative diagnostics, and sometimes perform corrective actions.

4.6.2 The Manager Process

A manager process can control the dynamic composition of the distributed application. RDS enables the manager to replace the definition of any observation operator and to instantiate and kill processes as instances of the objects. An authorized manager process may change the definition and configuration of the health application. For instance, it may modify the relative weights of the observers' evaluations in the overall health score, or change the evaluation algorithm. A manager process may configure which observer threads will report to a given HDMPI, which values and how often, and what is the relative weight of each measure.

Presentation of Information

A manager process receives information generated by the delegated agents. The application uses a graphical display to present simple metrics that represent the conclusion of many observations. These metrics are presented via generic graphical display objects, such as colored gauges or scales. Reports of utilization are displayed as gauges using colored ranges to classify information. The value of the health index

for a given LAN is classified into colored areas, e.g., unknown (blue), normal (green), warning (yellow), and problem (red), to indicate distinct operational conditions. Presentation objects implement additional features to call the attention of the human operator, like blinking text or beeping sounds.

4.6.3 The HDMPI

The HDMPI implements the main abstraction of the health application. For each observer, the HDMPI keeps configuration parameters, which are also modifiable by the manager. A manager may order the HDMPI to take into account an observer's report for the calculation of the health value or to ignore it. Observer threads provide the HDMPI with measurements derived from the computed sample observations, i.e., the f_i. The HDMPI uses these measurements to compute a numerical health score. An additional table of weights is used to compute the relative value score for a given measure compared with the other metrics. For example, if a certain rate has changed by x, then its contribution to the overall score will be given by a value $g(weights[f(x)])$, where f and g are functions defined by the HDMPI. Fine tuning $\vec{w}$ and the f_i is an interactive process.

For each observer, the HDMPI can keep an history log of the last n values received from it. When it receives a new value from an observer, it decides how much to affect the health score depending on which range the score is in, the percentage of change compared with the sensitivity, and the directionality of change. HDMPI performs a scalar product using $\vec{w}$ and $f_i(\vec{x})$. When the HDMPI receives a report from an observer it recomputes the health score as follows. First, it checks if the observer report should be ignored or not. If it is not ignored, the HDMPI establishes the range of the report, and calculates the difference with the previous value and its direction of change. It will then use the above information and apply the appropriate $g(weights[f(i)])$ to recalculate the health score.

Communication with the Manager

The HDMPI may report its evaluated score (1) by answering an explicit request, (2) by setting a private MIB variable, or (3) as event reports at a frequency established by the manager. A manager can invoke commands to influence the HDMPI computation. For example, it can get/set the current health value or any value of a observer, suspend/resume reporting at a given frequency, and add and delete observers.

4.6.4 Generic Observers

Generic observers execute as delegated threads under the control of an M_bD-server in close proximity to the networked elements being monitored. Observers collect raw data from the network elements and produce measurements which are reported to the HDMPI and/or the manager. Generic observers provide services which can be invoked remotely by manager processes to support a class of configuration

changes. Examples of these are the following: The list of processes that it should report to, device information specific for a given source of information, (e.g., for an SNMP agent, host and community), the length of the sampling intervals, the length of the history log, whether to forward messages to the manager (blind copies), and the frequency of event reports to manager. On instantiation, a generic object instance reports its configuration options and their current settings. During execution they accept manager requests to change their internal configuration.

Each observer defines the minimum relative sensitivity of variation in measures that can affect the health index. Sample observation changes which are below the sensitivity threshold are ignored. An observer may divide the space of its monitoring values into different ranges, each requiring different handling. For instance, an observer may define thresholds for LAN error rates which define ranges like "safe", "warning" and "dangerous". Each observer's report has a positive or negative contribution, e.g., when there is an increase (decrease) in the value received, the health score should also increase (decrease). For example, error rate has negative logic: the higher the rate, the worse the evaluated health value.

Algorithm for a Generic Observer

At initialization, an observer reads its own configuration, and communicates it to a manager process. It will obtain data values at the specified intervals, calculate differences with previous values, and apply some locally defined function, e.g., $g(weights[f(i)])$ to calculate its score. If the calculated value exceeds a triggering threshold, a report is sent to the health object and/or to the manager, and is also logged. If some corrective actions were defined, then the observer will execute them.

Each of these observers computes either a specific numeric score or a set of scores, according to the instructions received from the manager. For example, the operational state observer will obtain state information about a given list of network elements, including servers, workstations, PCs, routers, and so forth. The observer's processing will take into consideration the relative worth of each element, as defined by the manager. For example, the fact that a PC is not working may be normal whenever the user turns it off, so its relative importance is minor. However, if a file server is not operational, then the score evaluated by the state observer should change significantly, to reflect its relative significance to the well-being of the distributed system.

Event Reports and Corrective Actions

Event reports and alarm conditions may be raised by an observer when they evaluate an expression. Such expressions could be simple, e.g., the index score evaluated by an observer reaches below (above) a specified threshold. But they can also express complex relationships between measurements. For example,

```
if (((SomeCondition AND Down(Device1))) OR (P(Device1) < Q(t))) ....
```

Such conditions and thresholds can be defined as part of the generic object, can be

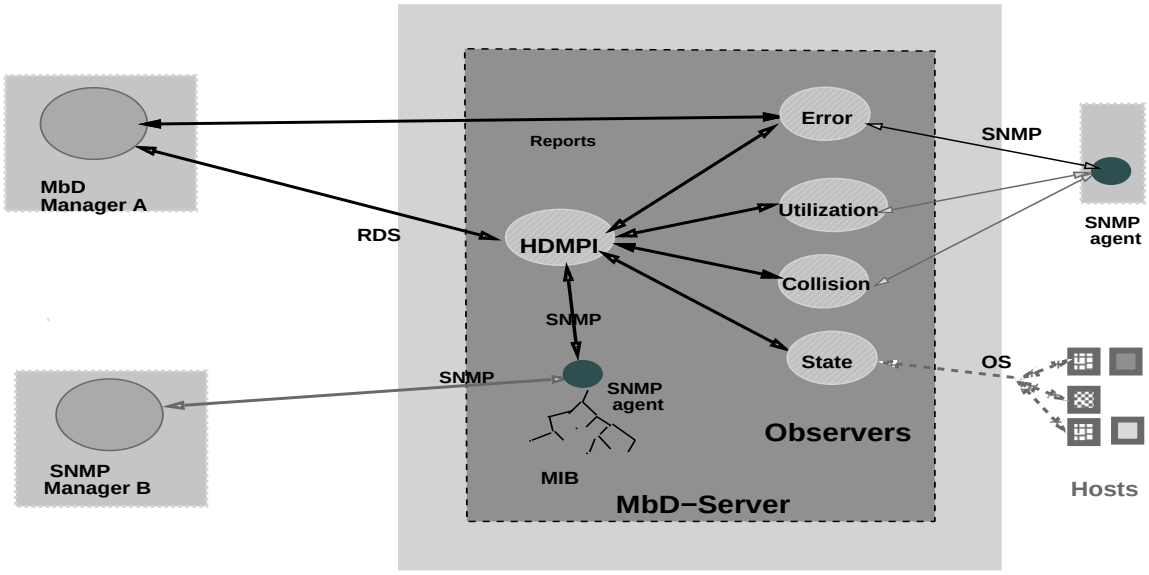

Figure 4.5: Prototype of Health Application

time dependent and dynamically adjustable. Thus, they can take into consideration network, behaviors for different periods of time.

Diagnostic procedures and corrective actions may be triggered as a result of an event. For example, if the level of an Ethernet utilization becomes too high, an observer thread may be instructed to perform some corrective action. It may identify the device which is the source of the largest amount of packets, and if possible, it can temporarily disconnect it from the network, until the load on the LAN returns to a normal range.

4.6.5 Prototype Implementation

We implemented a collection of generic objects to evaluate the health of an Ethernet LAN. We used an early prototype of the health application to demonstrate M_bD at the Synoptics booth at the InterOp 1991 Conference. The value of this function is computed based on several inputs about devices and LAN behavior: (a) Status of devices, (b) LAN Utilization, (c) Collision Rate, and (d) Error Rate. Figure 4.5 depicts the relationships between the components of the health application.

Observation Operators

The prototype implementation uses the following observation agents:

1. **Operational state of networked elements.** The *State* observer checks that each of the network elements is operating. Device down represents the situation when no data can be sent or received from the device. It is not possible to

determine if a device is really up unless the device supports SNMP or ping. For devices that support SNMP, the status can be obtained via the `ifOperStatus` (current) and `ifAdminStatus` (desired) for each interface. Alternatively, the status is obtained by using ICMP ECHO, (i.e., *"ping"*) for each IP device. Figure 4.6 shows a skeleton algorithm that evaluates the device status.

2. **Network Utilization.** This observer evaluates the ratio between the number of bytes received in some period of time and the maximum utilization possible for the given network. This information can be obtained from standard MIB-II variables, or via alternative methods, e.g., NNStat [Braden and Schon, 1991]. For the InterOp demo, we observed utilization at time t using the `s3EnetConcRxOk` from the private Synoptics MIB [Synoptics, 1990]

$$Utilization(t) = \frac{\texttt{s3EnetConcRxOk(t)} - \texttt{s3EnetConcRxOk}(t_0)}{(t - t_0) * 10000000}.$$

3. **Collision rate.** The collision rate observer will measure Ethernet collision counters provided by a private SNMP MIB [Synoptics, 1990]. The collision rate is computed as the number of collisions by the total number of packets over the same time period, again using counters from the private Synoptics MIB[4]:

$$Collisionrate(t) = \frac{\Delta(\texttt{s3EnetConcColls})}{\Delta(\texttt{s3EnetConcFrmsRxOk})}.$$

4. **Error rate.** A generic error observer may use SNMP MIB-II variables to measure error rates at the interface level and for specific types of traffic, e.g., IP, TCP, etc. Notice in Figure 4.5 that the error rate observer can send direct error report messages to the manager process using RDS_SendMessage(). At the InterOp demo, the Ethernet error rate was computed by adding CRC and misalignment errors from the private Synoptics MIB:

$$Errorrate(t) = \frac{\Delta(\texttt{s3EnetConcFCSErrors} + \texttt{s3EnetConcAlignErrors})}{\Delta(\texttt{s3EnetConcFrmsRxOk})}$$

Health Algorithm

Based on the experience of the SynOptics network engineers, we defined several algorithms for computing the health score. The following rules are an example of one of the early versions of the the health algorithm. A "perfect" LAN had a health score of 10. An alarm condition was raised when the health score reached 3. Conditions that changed the score included: -1 for every device down. -1 for every 5% average utilization above 30%. -2 if the peak utilization exceeds 50%. -2 for the loss of any

[4] $\Delta(F)$ means $F(t) - F(t_0)$.

```
IF DeviceType = NonTCP THEN
     IF s3EnetPortPartStatus is ENABLED (port for device, e.g. PC)
          AND s3EnetPortJabberStatus is OK
          AND s3EnetPortLinkStatus is DOWN THEN
              DeviceStatus = DOWN ELSE
                DeviceStatus = UNKNOWN

 IF DeviceType = Server THEN
      IF ICMP Ping = OK THEN
                DeviceStatus = UP ELSE
                DeviceStatus = DOWN

 IF DeviceType = SNMP THEN
      IF get_ipaddress = OK THEN
                DeviceStatus = UP ELSE
                DeviceStatus = DOWN
```

Figure 4.6: Device Status Algorithm

file server, since many users can not work without them. -1 (-2) If the collision rate is between 1-2% (> 2%). -1 if the error rate is above .2%. The HDMPI uses SNMP to store the computed index values on the local MIB. Thus, a remote SNMP manager like B can access the computed scores (see Figure 4.5).

Manager Presentation

The management console displayed several graphs of the computed observations: Health score, LAN utilization, Collision Rate, and Error Rate. Additional measures included the load on the file server host, the bridges, and the router. The health application also provided a configuration monitor which turned a console icon to yellow to signal a potential problem. For instance, the icon changed color if the health of the LAN fell below 3 anytime within one hour after a configuration change (adding a new device or upgrading its software) was made. A light started to blink at the console under any of the following conditions: (1) Average Utilization of 55%, (2) 3 devices and the server are down, (3) the server is down and utilization is 45%, (4) Collision rate is greater than 2% and error rate is above .2%.

Overview of Demo

A diagram of the network configuration at InterOp is given in Figure 4.7. An algorithm as defined above for the health of the LAN was delegated from a workstation to the *Network Control Engine* (NCE), a SPARC CPU at the hub. A PC executed

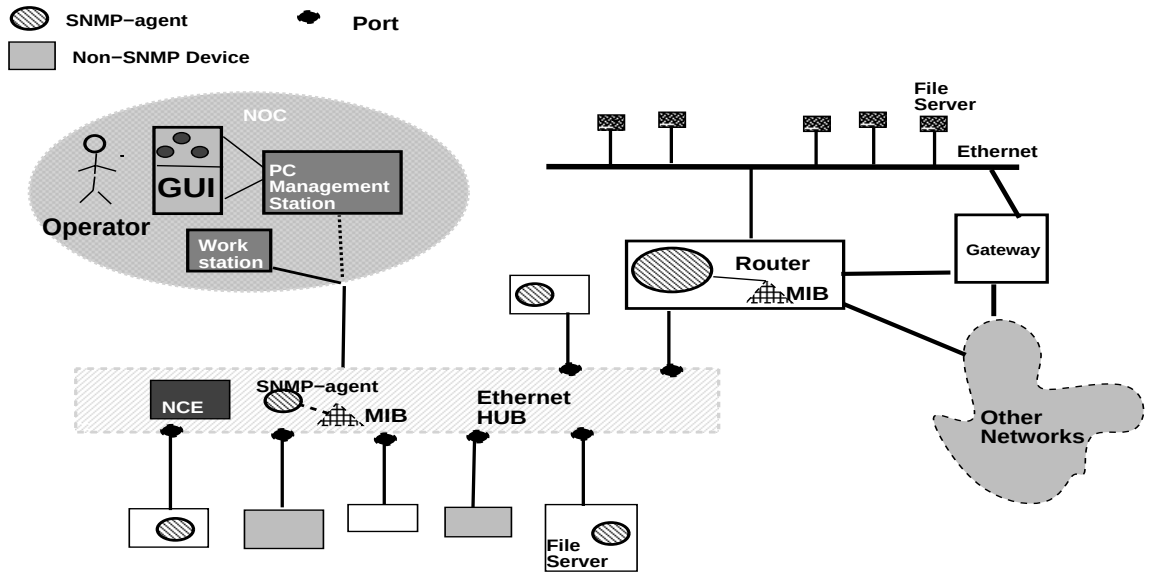

Figure 4.7: Synoptics Network at InterOp

a Synoptics management application that displayed the health meter and the other graphs. This application polled the Health values (via SNMP Gets) from the NCE at the hub. We defined a private SNMP MIB object Health, which was made available to any other SNMP management application, to demonstrate interoperability.

Initially the LAN health score was a perfect 10. We then disconnected a few devices and loaded the network using a sniffer. This resulted in the meter going down to below 3 and the generation of an alarm condition. We added different devices and brought the servers up and down by disconnecting their cables. One of the devices was sending too many CRC errors. This raised the error rate and the health score went below 3. The application automatically diagnosed and isolated the errors to the particular device and it partitioned the corresponding port. This brought the health of the LAN back to normal, and demonstrated how the Health application provided pro-active fault isolation.

We then demonstrated that some aspect of the network has changed. For instance, we (1) changed the score associated with the loss of the file server, (2) changed the health score algorithm to be sensitive to the time of the day, (3) dynamically changed the definition of error rates to include *"Runt"* packets, and (4) tuned the sampling intervals for the observations. These examples required the modification of the original health algorithm and communications between the delegated agents. We edited the changes at a SUN workstation and delegated them to the M_bD-server at the NCE. We tested the new health algorithm concurrently with the old one, suspending and resuming their execution as required. The new delegated agents executed as expected and the PC application did not notice any change.

4.7 Conclusions

Current network management paradigms do not support the temporal distribution and spatial decentralization required to compute real-time management functions effectively. We introduced a formal notation to describe the behaviors of managed entities and their observations by MIBs and by management applications. Management applications can dynamically compose operators to observe the behavior of the network elements. The output of these observation operators is used to reach management decisions. The traditional network management paradigm introduces many problems that prevent the effective computation of such operators.

We defined index functions to compress real time management observations at the location of the managed entities. We implemented an M_bD application that computes a health index of a network, and demonstrated its functionality at several conferences, including InterOp. The application performs a proactive diagnosis of failures and performs corrective actions in real-time. Management decisions, such as to temporarily disconnect a device, are executed efficiently, without the need for remote NOC intervention. Real-time operational data is effectively compressed at the M_bD-server, reducing the management data overhead on the network. The M_bD environment allowed network managers to tailor and customize the health application during execution.

5

MIB Views

5.1 Introduction

Management applications need to compute useful information from raw data collected in MIBs. Often such computations cannot be accomplished through remote interactions between the application and SNMP-agents. For example, suppose that an application needs to perform some analysis on all the routing table entries of a router. The application can use SNMP's `get-next` requests to retrive the routing table entries, one row at a time. This interaction, however, does not provide an atomic snapshot of the routing table at a given time, which is needed for consistency analysis. Instead, different sections of the table as seen by the application will reflect different versions of the routing table at different times.

This chapter introduces technologies to support extensibility of SNMP-agents to compute information from raw MIB data. It introduces an MIB View Definition Language (VDL) to specify computations over MIBs. Applications can delegate views defined in VDL to SNMP-agents in order to extract information of interest. For example, the routing analysis application could delegate views that take atomic snapshots of MIB tables. It could then retrieve the entries of these atomic snapshots using standard SNMP `get-next` requests.

In what follows we first describe several typical scenarios where applications must compute information from raw MIB data and cannot accomplish it effectively within the context of non-elastic SNMP agents. We then summarize the approach that we pursued to address these problems.

5.1.1 Examples of MIB Computations

Example: Filtering MIB Data

Applications often require means to retrieve selective information from MIB tables. Consider a diagnostic application of an ATM network that needs to detect problems arising at virtual circuits (VC). At present this application must retrieve

the entire VC table from every ATM switch in the network. It can then scan the respective operations statistics of each VC and decide whether it has a problem or not. With VC tables having potentially many thousands of entries, frequent retrievals of entire tables to detect potential problems of a few VCs is highly inefficient. Instead, it would be desirable to support selective retrieval whereby only entries that meet some filtering criterion (e.g., VCs whose operational statistics indicate problems) are retrieved.

This means that the application should be able to designate a filtering computation and pass it to the device and that the device can compute this filter before delivering data to the application. CMIP incorporated such filtering capabilities as part of the protocol[1]. Our system supports the computations of selective retrieval from SNMP-agents without modifying the SNMP protocol.

Example: Joining MIB Tables

A management application may need to quickly retrieve information that is scattered among several MIB tables, e.g., routing information specific to certain type of interfaces and their current utilization. Again, the SNMP protocol interaction requires the application to retrieve data from all the potential MIB tables to the platform host and only there to locally select the relevant data. For instance, the MIB-II `ipRouteTable` keeps track of IP routes, and interface information is found at the MIB-II `ifTable`. An application may need to correlate routes with interface utilization for capacity planning purposes. Such an application may need to retrieve data from the `ipRouteTable` and use the `ipRouteIfIndex` column as an index for the corresponding retrievals from the `ifTable`. Such an application may also need to impose a selection criterion on the retrieved values of the correlated entries. For instance, it may only need to retrieve `ipRouteTable` entries that have been learned via a specific mechanism, like the Border Gateway Protocol (BGP). In this case the selected entries will have `bgp` as the value of their `ipRouteProto` attribute.

Using SNMP, an application must retrieve individual columnar objects from the MIB, and then locally select the appropriate values and discard those that do not meet the selection criteria. Such management applications are forced to retrieve large amounts of data to the platform host to perform simple operations like filtering and joining MIB tables. This construction of an application-generated data models suffers from all the penalties associated with SNMP polling. Again, the SNMP paradigm does not provide any effective mechanisms to define such an external level user view out of its monolithic MIBs.

Example: Intrusion Attempt

Assume that a warning indicating intrusion attempts via the `finger` service has been received at the network operating center of a large organization. A net-

[1]See Appendix A.5 for a description of CMIP.

work manager may want to retrieve the information available about such remote TCP connections to all its hosts. The `tcpConnTable` table of MIB-II lists all the TCP connections of a host. The `tcpConnLocalPort` field of each entry row of the table indicates the local port of the connection. A manager may use these sample observations to monitor for connections to the `finger` service port, 79.

Notice that large hosts may have hundreds of TCP connections. Using SNMP, a management application must retrieve all the entries of the table, and locally select those that match the criteria, e.g., `tcpConnLocalPort` = 79. This retrieval will require many remote network interactions, and therefore will be very slow. Also, the retrieved table row variables may be concurrently updated before the entire retrieval has been completed. For instance, an intruder may use an ephemeral TCP connection to the `finger` port, and then use another port to continue its intrusion. In general, race conditions between concurrent updates of MIB data and their remote retrieval may result in management observations missing critical elements of the corresponding sample behavior sequence.

Example: Atomicity of Management Actions

Consider a router whose routing table has several thousands of entries. A management application (A) may need to retrieve all the entries of the table at once in order to analyze routing configuration patterns. Using SNMP, this retrieval will require many network interactions, and therefore will be very slow. More importantly, the retrieved variables may be updated before the entire retrieval has been completed. For instance, another management application, B, may concurrently update routing entries. This race condition between A and B can result in A being misled by the retrieved data.

This example shows that we need a way to make MIB computations with certain semantical guarantees. For instance, in the above example, A would need to specify an *"atomic snapshot"*. In such a snapshot the values of the MIB variables being retrieved are guaranteed not to change. Large table retrievals in SNMP do **not** have *atomic transaction* semantics. Each `Get` and `Get-next` exchange retrieves the current data values of each table entry at the time the request is serviced. Retrievals of large tables involve many SNMP `Get` requests. Yet the values stored in the MIB may change during the retrieval process. If this occurs, the table images at the management client side will be inconsistent with the values stored by the MIB.

5.1.2 MIBs Lack External Data Models

The above examples illustrate some of the problems that result from basing management observations on remote MIB access. These examples show that there is a need for mechanisms to perform MIB computations at the devices, while using standard data access protocols (e.g., SNMP) to these computations. A central difficulty in developing management applications is the need to bridge the gap between two different data models. Each management application defines its own structure for

the data it needs, but these structures are not identical to the MIB data structures. Management applications need to bridge the semantic gap between the conceptual-level data model rigidly defined in the MIB structures and the data model that they need.

Standard network management frameworks provide no support for management applications to dynamically define external data models as part of the MIBs. Although it is possible for applications to retrieve raw MIB data and compute the appropriate data model at the platform host, this is highly inefficient. Therefore, management applications are forced to retrieve large amounts of data to the platform to perform simple operations such as filtering. MIBs do not support the external definition of local computations that could focus their observation operators on relevant data. For example, they do not support filtering of MIB variables, joining of MIB tables, and atomic actions at the devices.

Reuse of Computations

Moreover, the SNMP paradigm does not provide means for various applications that execute at multiple management stations to share and reuse computations which are not part of the MIB. An application may compute a useful table of objects after filtering and correlating MIB data to perform some configuration analysis. Other manager applications could benefit from these computations, but they have no simple standard way to access them. Hence, each application will need to recompute them anew.

Different applications, and sometimes even the same application, may need to recreate the same external-level model several times from scratch. The lack of an external-level data model repository results in excessive and redundant retrievals and recomputations. In a multi-manager environment, particularly one that crosses several administrative domains, sharing of external data models can be very useful. For instance, a remote manager may have less bandwidth available and longer delays to recreate an external view. In such a case, access to an existing view could be the only effective way to access the data of the MIB.

Atomicity of Management Actions

Furthermore, in a multi-manager environment, it is difficult to ensure atomicity or transaction semantics of management actions over MIBs. Using SNMP, an action is invoked as a side-effect of a `Set` operation. When an action is invoked by setting a certain value to an object (i.e., a trigger object), an agent may treat one or more objects as parameters related to the action (parameter objects). But a parameter object set by one manager application may be modified by another application *before* the previous one invokes the action by setting the trigger object. This can lead to unintended and incorrect behaviors. The problem of computing a join of a table as an atomic action commonly occurs in many other network management scenarios. For example, resolution of routing problems typically involves correlation of routing,

address translation, and other configuration tables. It would be thus very useful to support effective computations of atomic joins.

Database Views

Traditional database systems support three data definition levels: internal, conceptual, and external [Elmasri and Navathe, 1989]. External data definitions allow remote applications to define *"views"* which are computations over the conceptual level data. Our approach is to support a similar mechanism over an MIB. Implementing a full fledged database system on top of an MIB would be too costly and complex for this purpose. Furthermore, the required semantics of these data models are somewhat different from those of traditional databases. The computations over MIBs are based on real-time sample observations over fast changing data. For this type of data there is usually no need to incur the computational costs and stable storage involved in ensuring transaction semantics. Such costs may be beyond the resource limitations of typical network devices. Our aim is, therefore, to design an effective external data model for MIBs without incurring the costs of a full-fledged database system.

5.1.3 Problems and Solutions

The main problem that this chapter addresses is how to support the dynamic definition of external data models for MIBs. There are several technical challenges that must be addressed. First, the new external data models need to be formally specified in an appropriate language. Specifications of external data models need to be translated into executable code that implements their semantics. Second, we need to efficiently bind the resulting code and data structures so that they have direct access to the original MIB data structures. Third, we need to implement mechanisms at the devices that ensure that the required semantics (e.g., atomicity) are enforced. Fourth, the new framework must be properly integrated within the standard management framework, so that existing applications and systems can interoperate with it.

MIB View Computations

Our approach is based on the M_bD paradigm and consists of (1) a *View Definition Language*, VDL, to specify MIB external views and (2) SNMP-agent extensions that implement them. External views are computed via instrumentations over an MIB, and are performed by a special M_bD-server. The results of these computations are organized as MIB variables that can be accessed by standard applications via SNMP. For example, MIB views can be used to support: (1) filtered retrieval of MIB objects, (2) relational joins between MIB tables, (3) access control to an MIB, and (4) atomic MIB snapshots.

Views may be used to support concurrent atomic actions in a multi-manager environment. A *view* can be delegated to an M_bD-server to define an action trigger and its parameters as an atomic group. For example, it can associate a queue of

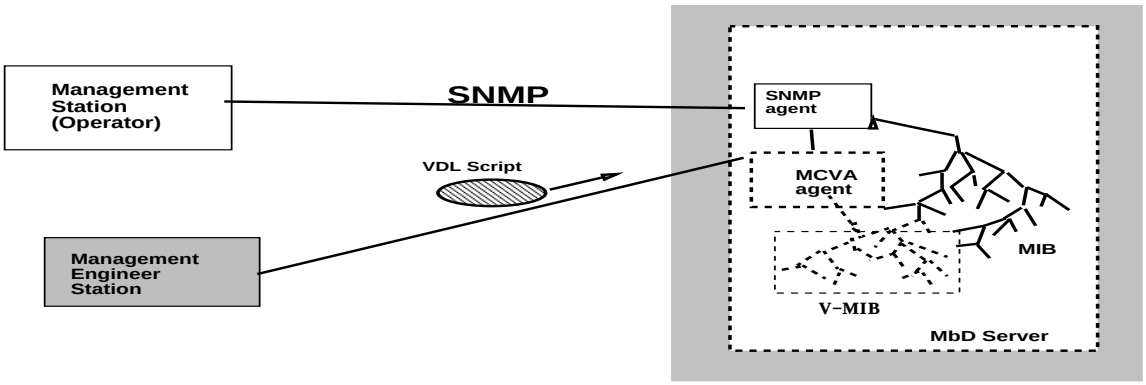

Figure 5.1: The MIB Computations System

action requests with the group. Each `Set` invoked by a manager to any object in the group will be queued. When all object `Set` requests by a given manager have been received in the queue, the action is invoked atomically. Should two managers access the action concurrently, their actions are serialized.

Chapter Organization

- Section 5.2 outlines the MIB Computations System, including its VDL language, and the M_bD-server runtime extensions.
- Section 5.3 presents detailed examples of how to use this framework to achieve filtering, joining of MIB tables, access control, and atomic snapshots.
- Section 5.4 describes MIB actions.
- Section 5.5 discusses the advantages of using M_bD to define and perform MIB computations, and provides a brief comparison with related work.
- Section 5.6 concludes the chapter.

5.2 The MIB Computations System

The MIB Computations System is depicted in Figure 5.1. It consists of a language to specify MIB views computations and agent extensions that implement these computations. The specifications of MIB computations are prescribed by a management *engineer*. These computations define the objects of a new *Virtual* MIB, V-MIB. The values of the V-MIB variables are computed by instrumentations over the original MIB. These computations are performed by a special M_bD-server, which executes in the same host or at close proximity to the original MIB. This M_bD-server supports a specialized OCP, namely, the MIB *Computations of Views Agent*, MCVA.

Management Information Bases

An MIB can be viewed as a simple database. Indeed, a *database* is defined by [Elmasri and Navathe, 1989] as a logically coherent collection of data with some inherent meaning, which is designed, built, and populated with data for a specific purpose. Thus, SNMP is the query language, the SMI is its *Data Definition Language* (DDL), and an SNMP-agent is a primitive *database management system*. A database system includes an intrinsic data model schema, that defines the structure of the database at three levels: internal, conceptual, and external. The *internal* level describes the physical storage of the data, e.g., the internal representation of MIB objects by an SNMP-agent. The *conceptual* level describes the entities, using the SMI language. The *external* level includes *external schemas* or *user views* which describe the parts of the database in which a particular user or group is interested, and hides the rest of the database.

In SNMP the external level is *identical* to the conceptual level. That is, there are no high-level data models geared toward specific applications. In SNMP the conceptual model defined by each MIB is the only external model. Management applications may need to perform analysis on data which is logically organized in different ways than those prescribed by the MIB. In database systems, different external levels are specified in *views*. A database view is a single table (relation) that is derived from other tables which can be either primitive base tables or previously defined views. For example, a view mapping could be used for selecting specific rows and columns from several tables, combining and correlating MIB data. SNMP management applications must retrieve "raw" MIB data and recreate an appropriate data model at the central host. As described in earlier chapters, this is inefficient and unscalable.

Management applications must bridge the gap between the data schemas rigidly defined by the MIB structures and the data model required by each application. MIB views address other problems that result from SNMP's lack of primitives to perform atomic actions and access control at the appropriate granularity. Also, some of these problems are exacerbated because SNMP is implemented over an unreliable transport mechanism, UDP, which may lose, duplicate, or deliver out of order any SNMP PDUs. We define a model of MIB computations that addresses these functional shortcomings of the SNMP paradigm.

We address these problems by using M_bD extensions to SNMP agents together with a specification language for computations over MIBs. This scheme allows management engineers to define arbitrary MIB computations. These computations can then be accessed by management stations, using SNMP queries. *Network management engineers* can create external level data models at the managed node side. These models are defined in the VDL language as MIB view objects and actions. MIB views are delegated agents that are translated by the M_bD-server into data structures and semantic routines that extend the MIB. This process is shown in Figure 5.2.

VDL constructs support the definition of selected table objects which are defined as joined MIB tables. It also supports selective retrievals of MIB objects that meet a selection criterion, based on arithmetic and logical operators. The M_bD-server

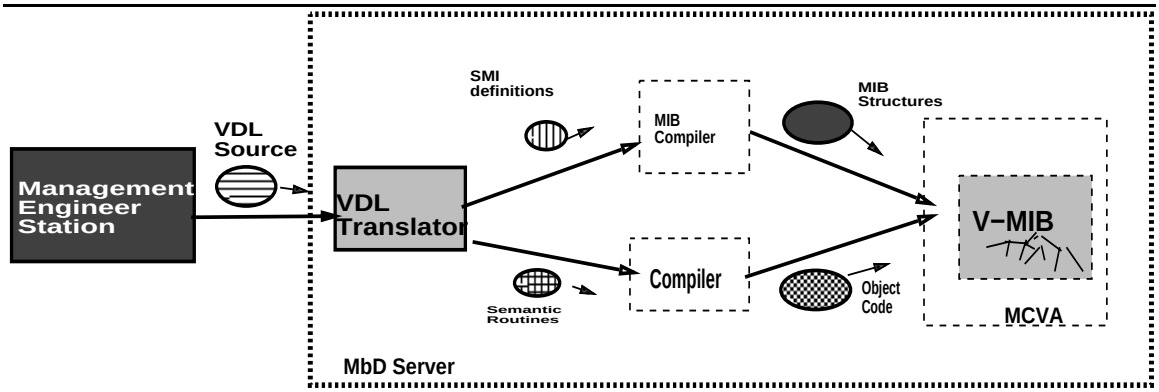

Figure 5.2: The Compilation of VDL definitions.

incorporates a translator that can accept or reject view definitions that do not conform to the VDL language or contain erroneous references to MIB objects. The translator compiles legal VDL code fragments, and generates the required MIB structures to hold the computed MIB values. The MCVA implements a dynamically extensible, virtual MIB that supports views and actions.

5.2.1 VDL - View Definition Language

VDL is a management language that provides constructs for defining MIB view objects and actions. VDL views are based on a restricted subset of SQL. SQL has appropriate constructs for view definition and is widely used for data management tasks. A VDL translator can accept or reject view definitions that do not conform to the VDL language or contain erroneous references to MIB objects.

Many database systems extend their data definition language (DDL) to support views. For SNMP MIBs, extending the DDL is akin to extending the SMI to support views, which is the the approach presented in [Arai and Yemini, 1995]. In contrast, our definition leaves the original SMI standard intact. We discuss some of the problems with that approach and compare the definitions of views in both languages in Section 5.5.2.

The syntax of our VDL is shown in Figure 5.3. It consists of constructs for creating view objects, (`CREATE MIBVIEW`), defining MIB actions (`CREATE MIBACTION`), and destroying them (`DELETE`).

The MIBVIEW construct

The CREATE MIBVIEW statement follows the pattern of the SQL CREATE VIEW command statement. The MIBVIEW gets a name (`<view-name>`), an optional list of attribute names for the columnar objects of the view (`<columnar-objects>`), and an optional attribute (`[SNAPSHOT]`) value for creating snapshot views. The table

```
CREATE MIBVIEW [SNAPSHOT] [<columnar-objects>] <view-name>
AS SELECT <attribute-list>
[FROM] <table-list>
[WHERE] <condition>
[DESCRIPTION] <description>


CREATE MIBACTION <action-name>
[ACTION] <action-procedure-identifier>
[IID]  <invocation-id>
[INPUT] <input-parameters>
[OUTPUT] <output-parameters>
[DESCRIPTION] <description>


DELETE <view-action-name>
```

Figure 5.3: VDL Syntax.

contents are defined using a select-from-where block construct.

`view-name` is the proposed object-name for the MIB view table. It must be unique in the scope of the MCVA, since it is used for identification purposes. If the view is accepted by the VDL translator, the MCVA assigns a corresponding SNMP Object IDentifier (OID) for the view table under the view tree. The OID is returned to the application which delegated the view.

`columnar-objects` are the new names of the individual columnar objects which constitute the conceptual rows of the new table.

`SNAPSHOT` is an optional attribute which specifies that the `MIBVIEW` should be computed once as an atomic snapshot of the MIB. A snapshot view maintains copies of the values of the objects at the time of the snapshot. The corresponding view table will have two extra fields: one of them is a snapshot-timestamp, which saves the value of the local date/time, and an instance index. A management application will be able to retrieve an existing snapshot view by specifying either the timestamp or the index. Each new instance of the snapshot view is computed when an appropriate SNMP query is received.

`attribute-list` is a list of attribute names whose values correspond to the elements (columnar objects) of the `MIBVIEW`. These typically correspond to columnar objects of the base tables or functions computed on their values.

`table-list` is a list of the MIB tables which need to be accessed in order to derive the `MIBVIEW` table[2].

`condition` is a Boolean expression that identifies the rows of the MIB tables which are to be retrieved as part of the `MIBVIEW`. If the `WHERE` clause is missing, there is no condition for row selection, so all rows are selected for the view. The Boolean expressions are built using the following Boolean operators: AND, OR, NOT, EQ ($=$), NEQ ($\neq$), LT ($<$), GT ($>$), LE ($\leq$), GE ($\geq$), with their obvious semantics.

`description` is a textual description to be appended in the `DESCRIPTION` field of the corresponding MIB table, for documentation purposes.

The MIBACTION construct

The definition of MIBACTION introduces a formal declaration of a procedural *interface* of a given action that is executed in the M$_b$D-server. The actual program that implements the action is usually a delegated agent, specified in any MSL. The MCVA maintains a table of actions, `vmib-action-table`, and for each action a table of invocations, `vmib-action-invocation-table`. The attributes of the MIBACTION provide a reference to the actual program, an identifier for each invocation, the input paramenters set by the manager at invocation time, and the output parameters set by the action as results.

`action-name` is the object name that identifies the action row in the `vmib-action-table` V-MIB table.

`action-procedure-identifier` specifies an OID which refers to the invocation "trigger" of the given action. For example, this OID can correspond to an MIB object that represents a delegated agent.

`invocation-id` is used as an identifier for the invocation, i.e., it identifies the conceptual row of the `vmib-action-invocation-table` V-MIB table for the invocation call.

`input-parameters` are the formal names of the parameters that need to be set by the management application prior to the execution of the action. The values of these input parameters are initialized atomically.

`output-parameters` are the formal names of the output parameters which the management script should set as results for the action.

[2]This may be redundant if the `attribute-list` identifies all the values.

5.2.2 The MIB Computations of Views Agent

The MCVA provides the runtime extensions that implement the external-level MIB views and actions. The MCVA executes as an OCP inside an M_bD-server configured with a VDL translator. To support the semantics of MIB views and actions, the MCVA binds the code generated by the VDL translator to compute the MIB views, to produce the values of the views columnar objects and actions. MCVA also enforces the atomicity of view snapshots by locking parts of the MIB.

MIB views provide external data schemas.

An MIB *view object* is a single MIB tabular object derived from other SNMP MIB tables. These tables can be either original MIB tables or previously defined view objects. A view object is a table that is needed by some management applications, but may not exist physically in the MIB. A view object maps the conceptual data model of the MIB to the external data model which is needed by the application. For example, a view mapping may involve selecting specific rows and columns from several tables, for the purpose of combining and correlating MIB data. Examples are provided in section 5.3.

The external data model created by a collection of MIB views is a *virtual* MIB, V-MIB. The values of the objects in V-MIB are derived from the values of existing MIB objects by applying a selection pattern, such as joins of MIB tables, upon a query.

Unfortunately, the updating of views which involve more than one base table is complicated and may even be ambiguous. In general, views defined on multiple tables using joins, and views defined using aggregate functions (e.g., arithmetic operators) are not updatable. Indeed, updating views is still an active research area [Elmasri and Navathe, 1989]. Thus, we apply views only for queries, i.e., read-access operations on MIBs.

MIB View Snapshots

View Snapshots are new V-MIB objects which provide an instantaneous copy of the values of a collection of MIB variables. The snapshot copy is executed "atomically" with respect to agent changes of the MIB values. That is, during the period of time that the snapshot is being taken, the agent will not update the MIB variables, neither by a manager `SetRequest` nor by reflecting changes in the state of the real managed objects. In other words, the corresponding MIB tables are effectively locked. This capability requires that the MCVA agent be able to effectively prevent MIB updates in the SNMP agent during the taking of the snapshot. The generated V-MIB objects provide a stored snapshot of the MIB state, which can then be retrieved via SNMP.

MIB Actions

MIB views can enforce atomic semantics of management actions over MIBs. For each action, there is a set of MIB objects that implement a *call frame*, which

includes the input parameters, the output parameters, and a call identifier. To ensure that there will be no race conditions between concurrent actions, each call frame is implemented as an entry row in a V-MIB table.

The MCVA guarantees that every invocation is independent of the others. An action is initiated by a manager retrieving an MIB object which is a "call counter" for the action. This call counter is a *test-and-increment* object. Everytime that a manager retrieves the counter, its value is incremented, thereby insuring that no two managers will get the same value.

To perform an action, a manager must set all the input parameters of a newly created conceptual row, and use a valid call counter value as the call identifier columnar object. When the `SetRequest` is received, the agent will allocate a new call-frame row indexed by the call counter, and invoke the action with those parameters. Upon completion of the action, the values of the output paramenters of the row are initialized, and a `GetResponse` is sent to the manager. The manager will then use a `GetRequest` to retrieve the output parameters.

MCVA integrates an SNMP-agent

The MCVA allows SNMP applications to access computed MIB views. Hence it must be closely integrated with an SNMP-agent. Note that there can be only one SNMP agent in any given host who listens to the corresponding UDP port 161. If the MCVA will execute on the same host as an existing SNMP agent, they need to cooperate.

Figure 5.4 describes an implementation of MCVA as part of an M_bD-server. In this design, there is a front end SNMP-agent that routes SNMP requests according to their OIDs. Requests meant for the original MIB are handled by a *"basic"* SNMP-agent component, while those whose OIDs relate to views are routed to the MCVA. The MCVA can directly invoke the methods that provide access to the internal SNMP- MIB data representations, and those of the MIB-views.

The front end of the SNMP agent forwards all SNMP requests for the V-MIB to the MCVA for proper handling. The MCVA must also be able to access the primary MIB contents for computing views and performing actions. Parts of this access could be accomplished by using SNMP. The advantage of using SNMP is that it does not require dealing with the internals of the SNMP-agent. However, the SNMP interface is insufficient, since the MCVA also needs to be able to lock the primitive SNMP-agent. Thus, some additional mechanism to access the SNMP-agent is required.

The responses that the MCVA generates can be sent directly to the appropriate manager, since they do not require handling by the SNMP-agent. This is possible since the SNMP standard mandates that the requests be received on port 161, but is silent concerning which port should be used by the agent to respond. For instance, if a manager uses its port 4123 to send request messages to the agent's 161 port, the agent can use its port 5678 to send a reply to the manager's 4123 port.

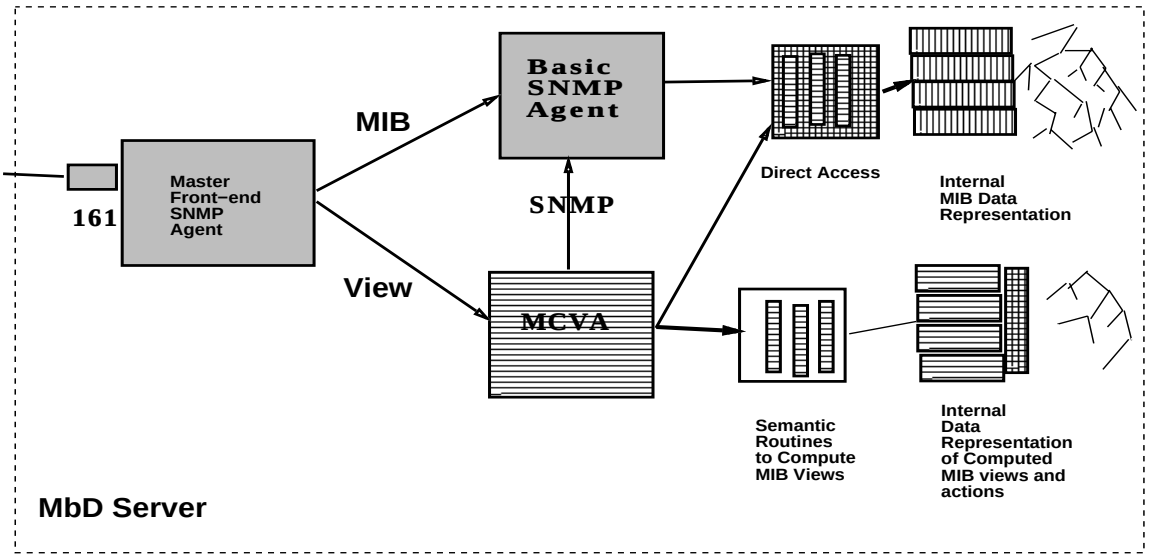

Figure 5.4: The Relationship between the MCVA and SNMP agents.

5.2.3 Applications can Learn about MIB Views

Management stations have different ways to learn about the MIB objects supported by any agent that they wish to query. A management application must know or find out whether or not a given agent supports a specific MIB, and in particular, the V-MIB. In general, management applications need to discover the agent MIB resources. This problem is somewhat orthogonal to our main concerns in this chapter, and can be solved in several different ways.

One way to solve the discovery problem is to provide an MIB of extended MIBs in the management agent itself. For instance, a *View-object-definitions*-MIB, is a repository mechanism built as an MIB View, which is a *table of contents* for the MIB. Whenever a new view is installed, the MCVA can add an entry into this table, with an appropriate description of the view. Management applications can query this MIB table to learn about the local view extensions of the V-MIB.

An alternative way is for the MCVA to deliver advertisements of the new MIB-views to a set of management stations. A management engineer can define this set as part of the M_bD-server configuration. This is only a partial solution, since only some management stations will receive the information. Moreover, this type of solution requires knowledge about the type of application and internal implementation details. For instance, it may require knowledge about how MIB objects are represented *internally* at any particular management station. Stations which are not listed, and/or belong to a different administrative domain would not receive the advertisement.

5.3 Applications of MIB Views

This section presents several examples that illustrate the use of MIB views to overcome many of the shortcomings of SNMP. These examples include:

- filtered retrieval of MIB objects,
- relational joins between MIB tables,
- access control to an MIB, and
- atomic MIB views.

5.3.1 MIB Views can filter MIB data

SNMP does not support value-based filtering of MIB data at its source. An MIB view object allows manager applications to filter table retrievals at their source, using arithmetic and Boolean operators over MIB values. Our method of filtering can be more efficient than the one pursued by CMIP, where filters are passed as part of the queries. VDL filters are delegated *ahead* of access. Therefore, they require no parsing and no interpretation during access time, as is the case for CMIP filters.

Simple Examples of Filters

Consider, for example, a typical routing device. The number of row entries in a routing table can be in the hundreds or even thousands. A management application may need to process all the entries in the table for which the `NextHop` field has certain values. For instance, the `NextHop` field may represent the IP address of a device which is due for service.

This selection criteria may be specified as *"retrieve all the* `ipRouteEntry` *rows for which the* `NextHop` *field is ..."*. For an SNMP application, this would require first retrieving all the rows of the routing table to the management station and then performing the data filtering at the station. Yet, only a few of these are the ones of interest to the management application. Thus, using SNMP involves a large overhead of requests.

Figure 5.5 presents an example of such a filter. The filter condition contains an MIB variable, `privateView.WantedRoute1`, which was defined for this purpose. A management application can set this variable to reuse the filter for different IP addresses. Similar filters can be defined for other large tables. For example, another useful filter could select TCP connections from the `tcpConnTable` by their local ports. Figure 5.6 defines a filter criterion that selects all interfaces for which the number of configured VPCs exceeds 2000, or the number of configured VCCs exceeds 30000.

```
CREATE MIBVIEW viewRouteNextHop1 (v2RD, v2RII, v2RNH, v2RA)
AS SELECT ipRouteDest, ipRouteIfIndex, ipRouteNextHop, ipRouteAge
FROM ipRouteTable
WHERE (ipRouteNextHop = privateView.WantedRoute1)
DESCRIPTION = ``Routes for NextHop WanterRoute1''
```

Figure 5.5: Routing Filter View

```
CREATE MIBVIEW viewvpcvccexcess1
          (v3II, v3Maxvpcs, v3ConfVpcs, v3ConfVccs)
AS SELECT ifIndex, atmInterfaceMaxVpcs,
          atmInterfaceConfVpcs, atmInterfaceConfVccs
FROM atmInterfaceConfTable
WHERE (atmInterfaceConfVpcs > 2000) OR (atmInterfaceConfVccs > 30000)
DESCRIPTION = ``Excess VPCs or VCCs filter''
```

Figure 5.6: ATM Filter View with Boolean expression.

A detailed example

MIB view objects may be used to select objects that meet filtering criteria of interest. The example in Figure 5.7 defines a view based on MIB-II interface table `ifTable`. This view selects three MIB attributes, `ifindex, ifSpeed, ifPhysAddress`, for all ATM interfaces (i.e., those whose interface type is 37). Without this filter view, a management application would have to retrieve all the interface table entries and then check the type of each one.

If the ratio of ATM interfaces to the total number of interfaces is small, the view object can save the management station several retrievals for each MIB. For instance, a management application which is doing an inventory of ATM interfaces in a large network must query many SNMP-agents. In such an environment the compounded savings can be very large.

The VDL translator takes the definition of `viewAtmIfTable1`, and produces an appropriate template in ASN.1 syntax, as shown on Figure 5.8. This template follows the SMI rules and can be used by any SNMP MIB compiler. An NOC application can retrieve the selected rows of the the view table via SNMP `GET` and `GET-NEXT`. For example, consider the following SNMP request using the above view defined over the sample interface table given in Figure 5.9.

```
GetNextRequest(v1ifIndex, v1ifSpeed, v1ifPhAdd)
```

```
CREATE MIBVIEW viewAtmIfTable1 (v1ifIndex, v1ifSpeed, v1ifPhAdd)
AS SELECT ifindex, ifSpeed, ifPhysAddress
FROM ifTable
WHERE ifType = 37
DESCRIPTION = "ATM Interface Table View from MIB-2 Table"
```

Figure 5.7: Example of VDL view for Interface Table.

The MCVA agent will scan the interface table for the appropriate next entry, i.e., the first ATM row in the interface table. It will then return the corresponding values in a `Get-Response` SNMP PDU.

```
GetResponse( (v1ifIndex.3 = 3), (v1ifSpeed.3 = 155520000),
             (v1.ifPhAdd.3 = 00000C03C82C))
```

5.3.2 Views Support Relational Joins of MIB Tables

Management applications often require information following a data model which is different than the one predefined by the MIB. The required data is contained in several different MIB tables and is selected based on some matching criteria. In relational database terms, an application may want to perform a join between several relations (tables).

For example, in order to isolate some types of faults, a management application may need to correlate the status of logical links (e.g., PPP links [Simpson, 1993b]) and the status of the physical ports which they use (e.g., HDLC [Simpson, 1993a]). This correlation can be useful to help applications diagnose and isolate faults that are manifested by managed objects associated with both layers. Such correlation could be accomplished by computing a join of the respective tables.

Management applications may also need to quickly retrieve information that is scattered among several tables, e.g., information specific to certain type of interfaces and their current utilization. For example, ATM interface information is found at the MIB-II `ifTable` and at the `atmInterfaceConfTable`. Using SNMP, an application will need to retrieve values from one table in order to use them as indices for retrievals from another table.

Again, the SNMP paradigm does not provide any mechanisms to define such an external level user view of the MIB data. Using SNMP, an application must retrieve individual columnar objects from the MIB and then create an appropriate user-level data model by itself. This construction of application-generated data models suffers from all the penalties associated with SNMP polling.

```
 viewAtmIfTable1 OBJECT-TYPE
    SYNTAX SEQUENCE OF vAtmIfTableEntry
    ACCESS not-accessible
    STATUS optional
    DESCRIPTION "ATM Interface Table View from MIB-2 Table"
    INDEX {v1IfIndex}
    ::= {mibview 1}

vAtmIfTableEntry1 OBJECT-TYPE
    SYNTAX VAtmIfTableEntry1
    ACCESS not-accessible
    STATUS optional
    ::= {viewAtmIfTable 1}

VAtmIfTableEntry1 ::= SEQUENCE {
    v1ifIndex INTEGER,
    v1ifSpeed Gauge,
    v1ifPhAdd PhysAddress
    }
```

Figure 5.8: SMI statements derived by VDL translator.

Example of Joining Tables

A better approach is to let the MCVA compute a join, and retrieve only the rows that belong to the joined table. This is particularly efficient when dealing with very large tables, when only a few of the relations will meet the filtering constraints of the join.

For example, the interfaces table defines a conceptual row (tuple) for each of the (`ifNumber`) physical interfaces of the managed entity. Each row in this table has 22 columnar objects (attributes) as shown in Figure 5.9:
`ifIndex (1), ifDescr (2), ifType (3), ···, ifSpecific (22)`
The ATM-MIB defines an `atmInterfaceConfTable` table, indexed by `ifIndex`, which contains one entry row per ATM interface port, with 12 columnar objects per entry:
`atmInterfaceMaxVpcs (1), atmInterfaceMaxVccs (2),`
`atmInterfaceConfVpcs (3), ···, atmInterfaceMyNeighborIfName (12)`

A join of two tables can impose a selection criterion. For example, the following condition specifies that only ATM interfaces which support ILMI[3] are selected for the join:

[3]ILMI is the Interim Local Management Interface, which allows the user-side and network-side of a User Network Interface to exchange information concerning a local ATM interface.

ifIndex	*ifDescr*	*ifType*	*ifMtu*	*ifSpeed*	*ifPhysAddress*	···	ifSpecific
1	Ethernet0	6	1500	10000000	00000C03C84C	···	···
2	Fddi0	15	4470	100000000	00000C03C42C	···	···
3	ATM1	37	53	155520000	00000C03C82C	···	···
4	ATM2	37	53	155520000	00000C058C03	···	···
5	ATM3	37	53	50112000	00000C058C05	···	···
6	Ethernet1	6	1500	10000000	00000C03C82A	···	···
7	Ethernet2	6	1500	10000000	00000C03C82B	···	···
8	Ethernet3	6	1500	10000000	00000C03C82D	···	···
9	Ethernet4	6	1500	10000000	00000C03C82E	···	···
10	Fddi1	15	4470	100000000	00000C03C82F	···	···
11	Fddi2	15	4470	100000000	00000C01AAAC	···	···
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 5.9: Sample Interface Table

`((atmInterfaceIlmiVpi` $\neq$ `0) OR (atmInterfaceIlmiVci` $\neq$ `0))`.
If the values of `atmInterfaceIlmiVpi` and `atmInterfaceIlmiVci` are both equal to zero then the ILMI is not supported at this ATM interface.

An application may want to access, at once, several attributes from each interface which are spread over both the interfaces table and the ATM table [M.Ahmed and K.Tesink, 1994]. For instance, a management application may want to retrieve for each ATM interface:
(1) the interface index,
(2) the total bandwidth in bits per second for use by the ATM layer,
(3) the maximum number of Virtual Path Connections supported,
(4) the maximum number of Virtual Channel Connections supported, and
(5) the IP address of the neighbor system connected to the far end of this interface, to which SNMP messages can be sent.

```
CREATE MIBVIEW viewAtmIfTable4
  (v4ii, v4is, v4aimvpcs, v4aimvccs, v4myia)
AS SELECT ifIndex, ifSpeed, atmInterfaceMaxVpcs,
          atmInterfaceMaxVccs, atmInterfaceMyNeighborIpAddress
FROM ifTable, atmInterfaceConfTable
WHERE (atmInterfaceIlmiVpi NEQ 0) OR (atmInterfaceIlmiVci NEQ 0))
DESCRIPTION=''Join of ATM and Interface tables that ''
```

Figure 5.10: Example of VDL statement to join 2 tables.

```
 viewAtmIfTable4 OBJECT-TYPE
    SYNTAX SEQUENCE OF vAtmIfTableEntry
    ACCESS not-accessible
    STATUS optional
    DESCRIPTION "Join of ATM and Interface tables''
    INDEX {vIfIndex}
    ::= {view 1}

vAtmIfTable4Entry OBJECT-TYPE
    SYNTAX VAtmIfTableEntry
    ACCESS not-accessible
    STATUS optional
    ::= {viewAtmIfTable4 1}

VAtmIfTable4Entry ::= SEQUENCE {
    v4ii INTEGER,
    v4is Gauge,
    v4aimvccs INTEGER,
    v4aimvccs INTEGER,
    v4myia IpAddress
    }
```

Figure 5.11: SMI statements derived from VDL statement to join 2 tables.

The MIB view-object defined in Figure 5.10 joins data from the MIB-II interface table, `ifTable`, and from the ATM MIB `atmInterfaceConfTable`. Both tables are indexed by the same index, `ifIndex`. From that definition, the VDL translator derives an appropriate template in ASN.1, as shown in Figure 5.11. This is an example of the very useful select-project-join queries. Figure 5.14 depicts an example of the joined table computed using the base tables depicted in Figure 5.12 and Figure 5.13.

5.3.3 MIB Access Control

A site may need to impose administrative access restrictions to prevent arbitrary remote managers from retrieving the original MIB tables. Consider a collection of *Private Virtual Networks* (PVNs) sharing a common underlying physical network, e.g., X.25, Frame Relay or ATM. Such PVNs are commonly used by telecommunication service providers to partition bandwidth among multiple organizations. Each service provider is responsible for managing the entire shared network as a whole. Customers can only view and manage their individual portions of the shared service. A network management application responsible for managing the various PVNs must share access to the SNMP-agents of the underlying network elements. However, these

ifIndex	*ifDescr*	*ifType*	*ifMtu*	*ifSpeed*	···
1	Ethernet0	ethernet(6)	1500	10000000	···
2	Fddi0	fddi(15)	4470	100000000	···
3	Ethernet2	ethernet(6)	1500	10000000	···
4	ATM1	atm(37)	53	155520000	···
5	ATM2	atm(37)	53	50112000	···
6	ATM3	atm(37)	53	1728000	···
7	ATM4	atm(37)	53	1728000	···
8	ATM5	atm(37)	53	50112000	···
9	Ethernet3	ethernet(6)	1500	10000000	···
10	Ethernet4	ethernet(6)	1500	10000000	···
11	Fddi1	fddi(15)	4470	100000000	···
12	Fddi2	fddi(15)	4470	100000000	···
⋮	⋮	⋮	⋮	⋮	⋮

Figure 5.12: Sample Interface Table

INDEX	*MaxVpcs*	*MaxVccs*	···	`IlmiVpi`	`IlmiVci`	···	*MyNeighborIpAddr*	···
4	200	10000	···	127	32000	···	128.59.16.1	···
5	300	20000	···	0	0	···	128.59.16.3	···
6	255	25000	···	64	0	···	128.59.18.12	···
7	200	15000	···	255	65535	···	128.59.19.10	···
8	300	20000	···	0	0	···	128.59.16.12	···
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 5.13: Sample ATM Interface Table

v4ii	*v4is*	*v4aimvpcs*	*v4aimvccs*	*v4myia*
4	155520000	200	10000	128.59.16.1
6	1728000	255	25000	128.59.18.12
7	1728000	300	20000	128.59.19.10
⋮	⋮	⋮	⋮	⋮

Figure 5.14: Sample Joined ATM Interface Table

elements are typically owned by the subscriber organizations.

For obvious privacy reasons, access by such an application should be limited to monitor and control only those resources that are required for specific management functions, e.g., fault diagnosis or accounting. A customer site may also require some access to global information to optimize its use of the available resources. For instance, an organization may want to access network load and pricing information in order to decide at what time to schedule some large data transfer. However, such global data may include information which is private to another organization. For instance, both organizations could be financial entities that share a common PVN at an overseas location. In many countries there is only one network service provider. It would be inappropriate for the PVN service provider to indirectly reveal confidential information of an organization to its competitors.

The original SNMP standard does not provide an adequate framework for this access control. Access control in SNMP is defined by a *community profile*, which consists of subsets of the MIB objects and their corresponding access modes (R/O or R/W). By defining several communities, an SNMP-agent can limit access to its MIB, and provide different levels of access to different management stations. Thus, access control is confined to those which the SNMP-agent has defined in its MIB. As conditions and configurations change, a PVN management system may need to retrieve different pieces of information, which are not subsets of the MIB, and which the local administrator could not have foresee when its community profiles were defined.

MIB Views Provide Access Control to MIB Data

MIB views can provide an effective mechanism to protect access to data. An MIB view can be used to define limited access granularity for remote management applications while maintaining access restrictions on the rest of the MIB. Thus MIB views can support management across multi-domain networks.

For example, a PVN management application may need to retrieve information related to TCP connections using a given port to discover intrusion attacks. For instance, the finger TCP port 79 [Zimmerman, 1991] has been used for many penetration attempts. Assume that a series of cross-domain attacks using the finger port have been reported. The information about TCP connections could be used by the PVN management application to assess whether or not the site is currently the target of a remote attack, and to assist in tracing the source of the attack. These views cannot be predefined statically at the agent, as the mode and types of intrusion attacks continuously evolve.

An MIB view could be defined by an authorized management engineer to retrieve the information related to TCP connections using a given port. This view would be delegated to all the relevant nodes, enabling quick polling to retrieve information concerning the source of the attacks. An example of such a view is defined in Figure 5.15 below. In this case the view retrieves the IP addresses and ports of the connections with local port 79.

```
CREATE MIBVIEW viewFingerPortConnections
    (v5State, v5LocalAddress, v5RemAddress, v5RemPort)
AS SELECT tcpConnState, tcpConnLocalAddress, tcpConnRemAddress, tcpConnRemPort
FROM tcpConnTable
WHERE (tcpConnLocalPort = 79)
DESCRIPTION=‘‘TCP Connections to Finger Port ’’
```

Figure 5.15: Access Control to a subset of the tcpConnTable.

5.3.4 Atomic MIB Views

A snapshot MIB view is computed once and thereafter maintains copies of the values of the objects at the time of the snapshot. Snapshot views are very useful to investigate transient problems of short duration. Some of these problems are often handled by automatic recovery mechanisms which quickly mask the symptoms of the underlying problem. For example, routing algorithms can dynamically adjust routes to react to changing network conditions. Algorithms like RIP's [Hedrick, 1988] *distance-vector algorithm*, for instance, try to find the best route for every destination. Thus, an intermittent routing problem may be masked by the routing algorithm itself.

A management application may need to retrieve all the IP routing tables of a router at once in order to analyze routing configuration patterns. Using SNMP, this retrieval will require many network interactions, and therefore will be very slow. More importantly, some of the retrieved variables may be updated before the retrieval has been completed. A management application would be misled by such concurrent updates. Management applications need a way to specify an atomic snapshot retrieval, in which the values of the MIB variables being retrieved are guaranteed not to change.

To detect such intermittent conditions, a manager may take several snapshots of the routing table at specified times, and later evaluate the differences between them. Each new instance of the snapshot view is computed when an appropriate SNMP query is received. An MCVA agent may build a snapshot of an IP routing table, e.g., `ipRouteTable`, for all entries whose route use a given collection of local interfaces as their next hop of the route. Thus, the manager application will be guaranteed that the values returned from quering the table snapshot are "consistent". The resulting view table will be timestamped to identify the snapshot. By taking several snapshots, (i.e., a trace of views) a management application can analyze more accurately the behavior of dynamic route changes. This example is shown in Figure 5.16.

5.4 MIB Actions

Management applications need to invoke imperative actions at managed entities. For example, remote actions can be used to:

```
CREATE MIBVIEW SNAPSHOT viewipRouteTable1
    (v6Dest, v6Ifindex, v6Proto, v6Age)
AS SELECT ipRouteDest, ipRouteIfindex, ipRouteProto, ipRouteAge
FROM ipRouteTable
WHERE (ipRouteIfindex = 1) OR (ipRouteIfindex = 3)
DESCRIPTION = ``Routing Snapshot''
```

Figure 5.16: Snapshot View of Routes via Interface 1 or 3.

- control the configuration of managed devices, e.g., to establish permanent virtual circuits through a switch,
- invoke diagnostic procedures when a malfunction is discovered, e.g., perform a hardware diagnostics program for a malfunctioning workstation, and
- perform corrective adjustments to address performance inefficiencies or failures, e.g., modify routing tables to compensate for a malfunctioning interface in a router.

SNMP provides no explicit mechanism for issuing a command to an agent to perform an action[4]. Rather, a predefined command can be invoked by an implicit mechanism, which involves changing the value of an MIB variable which has been specifically designated for this role. As a side effect of changing its value, the object will trigger the execution of the action. For instance, a *reboot* command can be implemented as an MIB private object, e.g., `reBoot`, which has a default value of 0. When a manager sets the `reBoot`'s value to 1, the managed system performs the action (i.e., reboots) and resets the object value to 0.

Many management actions, however, need to be invoked with parameters. Consider an action with n paramenters, $A(p_1, p_2, ...p_n)$. In SNMP, the value of MIB objects can be set in order to be used as parameters to the action. An SNMP-manager could invoke this action by setting the corresponding MIB variables A, p_1, ..., p_n in one `SetRequest` PDU. The problem of this approach is how to ensure atomicity of the invocation of actions. The standard specifies that all objects in a single `SetRequest` must be set *"as if simultaneously"*.

Note, however, that the *order* of the value assignments required by each `SetRequest` is arbitrarily defined by the agent. If a manager would use the same `SetRequest` to set both the parameters and the action triggering object, its call semantics could be undefined if the action is triggered with the wrong parameter values. There is no guarantee that the triggering will happen only after all the parameters (p_i) are set to their corresponding new values.

[4]In contrast, CMIP supports explicit invocation of remote procedure calls via the `M-ACTION` primitive.

Hence, a manager must ensure that all parameters are set prior to triggering the execution of an action that uses them. Thus, two `SetRequest` PDUs are needed for each action, the first PDU for setting all the parameters (p_i), and the second for actually triggering the action (A). This situation creates a potential race condition between managers. For instance, one manager could reset the values of parameters just set by another manager who then issues an action triggering request. Such interference could lead to erroneous results.

For example, consider a management action which allocates a buffer for some diagnostics results. Before invoking the action, a manager (A) must set the starting address of the desired buffer in an appropriate MIB object, `bufferStartingAddress`. A second manager (B) could change the value of `bufferStartingAddress` after A set it, but before the action was triggered. This would clearly result in erroneous results.

Unfortunately, such races are not rare, since SNMP is implemented over UDP, an unreliable transport mechanism. UDP may lose, duplicate or deliver out of order the SNMP `SetRequest` PDUs. The second `SetRequest`, issued to trigger the action, can not be sent before the first one has succeeded. Under those circumstances, a management application must delay the second PDU until it is certain that the packets carrying the first request no longer exist in the network [Rose, 1994]. This requires the manager to wait for the expiration of the `time-to-live` field of the IP datagrams used for the first `SetRequest`.

5.4.1 MCVA Supports Atomic MIB Actions

An MCVA agent can support atomic actions with parameters. Let us consider an example of an action to configure ATM connections. An ATM Virtual Connection (VC) is characterized by a traffic pattern, its Quality of Service (QoS), and its topology. The establishment of a VC consists of reserving appropriate Virtual Links (VLs), characterizing the traffic on the VLs, cross-connecting the VLs in Intermediate Systems (ISs), and associating the VLs with user applications. The definitions of ATM MIB objects are given in [M.Ahmed and K.Tesink, 1994]. An elaborate procedure that uses the ATM MIB to configure virtual connections (VCs) is described in [Tesink and Brunner, 1994]. The procedure involves the steps of the management script detailed in Figure 5.17 below.

The steps in the management script involve retrieving and setting several MIB objects and require careful error checking at each step. Micro-managing the steps of the procedure from an NOC host requires several SNMP exchanges over UDP. This exchange exposes the management procedure to race conditions with other managers. Furthermore, recall that UDP is an unreliable transport protocol, which can lose, duplicate or deliver the packets out of order.

A better alternative is to define an MIB action, associated with the corresponding management procedure. This procedure can perform all the steps at the MCVA which is adjacent to the ATM switch and the respective hosts. Such a pro-

```
Retrieve interface VC constraints from atmInterfaceConfTable.
Create a VL entry in the atmVclTable.
If successful
   increment the values of Vpcs and Vccs in for the interface.
Create a row in atmTrafficDescrParamTable and
   initialize its columnar objects.
Refer the atm[Vpl|Vcl][Receive|Transmit]TrafficDescrIndex
   in the atm[Vpl|Vcl]Table to the corresponding
   atmTrafficDescrParamTable rows.
Activate the VLs at each host and the IS.
Obtain a unique cross-connect index from atm[Vp|Vc]CrossConnectIndexNext.
Connect the VCLs by creating a row in atmVcCrossConnectTable.
If successful
   fill atmVcCrossConnectIndex in the atmVclTable
   activate the cross-connected VLs.
```

Figure 5.17: Script for Configuring ATM Connections.

cedure could be a implemented as a delegated agent to the corresponding MCVA. The delegated procedure is then assigned a triggering object in the V-MIB, say `vatmMbdActionConfigureVC1`. The definition of an MIB action view that could be used to invoke this procedure in VDL is given in Figure 5.18.

```
CREATE MIBACTION vatmConfigureVC
ACTION vatmMbdActionConfigureVC1
IID  vatmConfigureVCActionID
INPUT vatmIfIndex, vatmVclvpi, vatmVclVci,
      vatmTrafficDescrType, vatmTrafficDescrParam1
OUTPUT vatmVcCrossConnectIndex
```

Figure 5.18: VDL Specification for Action that configures ATM Connections.

After the action script and its interface definition have been delegated to the MCVA agent, a manager may invoke the action. An action is initiated by a manager retrieving the call counter from the V-MIB. Every time a manager retrieves the `viewAtomicCallCounter`, its value is incremented by the MCVA. To reduce the possibility of collisions, the MCVA may record an identifier of the manager to insure that the same one will execute the action. Thus, unless two managers actively cooperate to collide by sharing the values of `viewAtomicCallCounter`, this object assures no conflicts.

To perform the action, a manager must set all the input parameters of a newly created conceptual row. In our example, the table of calls is `vatmConfigureVC`, and the parameters that must be set are `vatmConfigureVCActionID, vatmIfIndex, vatmVclvpi, vatmVclVci, vatmTrafficDescrType,` and `vatmTrafficDescrParam1`. The value of `vatmConfigureVCActionID` must be a valid call counter value retrieved from `viewAtomicCallCounter`.

When such a `SetRequest` is received, MCVA will allocate a new call-frame row indexed by the call counter, and invoke the action with those parameters. While the action is in progress, the columnar value `action-state` will have the value `in-progress`. Upon completion of the action, the values of the output paramenters of the row are initialized, and `action-state` will have the value `completed`. At this time a response is sent to the manager that invoked the action. The manager will then use a `GetRequest` to retrieve the output parameters.

The semantics of the atomicity desired among concurrent actions can be explicitly encapsulated in the delegated program that implements the action. For example, the action may specify that only one instance of this type can execute concurrently in the MCVA, or that all actions can execute concurrently, or prescribe mutual-exclusion zones, semaphores, or other concurrency control mechanisms.

5.5 Using MbD to Implement MIB Computations

SNMP management applications need to bridge the gap between the data model rigidly defined in the MIB structures and the data model required by the application itself. Implementing this mapping complicates both the development and the performance of SNMP applications. Indeed, the lack of an appropriate external data model is one of the reasons for the dearth of applications that provide more functionality than simple browsing of MIB values.

5.5.1 Performance Considerations

The V-MIB extensions implement efficient computations that can deliver the required information to management applications via SNMP. It is more efficient to provide managers with access to computed MIB views than to have them continuously poll for the original predefined objects. SNMP "fundamentalists" may claim that the MCVA extensions will probably require a lot of computational power on the managed element. The MCVA functionality is organized in a multithreaded environment, e.g., an M_bD-server. Thus, it can be configured to execute only the required computations, according to the host device computational abilities. Views can actually save host resources which are spent in polling and creating external-level views by managers. The MCVA is not intended to execute in a resource poor environment which cannot afford or may not require this level of manageability. Devices which can barely afford a traditional SNMP agent will continue to support only the basic SNMP functionality, and MCVA agents may be able to proxy for them.

The response time for an SNMP query may be significantly different (sometimes faster, sometimes slower) for view-objects than for the original MIB values. For example, an SNMP query of a standard MIB object may require the SNMP-agent to perform a slow proxy operation to a remote agent. A similar query to a derived MIB will be answered much faster if the derived object has been computed in an earlier snapshot. On the other hand, a view-object may require extensive computing, e.g., table joins, thus having significantly slower response time than a comparable SNMP query that only retrieves the original tables.

Views and Actions

Views and actions are applied in different ways to deal with the shortcomings and deficiencies of the SNMP paradigm, while maintaining interoperability with SNMP-based applications. Applying these extensions does **not** require making changes to the core SNMP standards, i.e., the protocol and the SMI. Therefore, any SNMP manager application can access and take advantage of the computed MIB extensions. Providing derived objects such as MIB views at the "agent" side is somewhat contrary to the minimalist computational paradigm expected for SNMP-agents. In this *"simple"* paradigm, device resources are considered to be at a premium, and therefore all MIB objects are primitive and derivations should occur at the management station side. The MCVA is tightly integrated with the corresponding SNMP-agent inside an M_bD-server.

5.5.2 Related Work

Some related work has focused on a theoretical *modelling* of the semantics and behavior of managed networks. For example, two specification techniques that combine object-oriented and relation algebraic methods to model MIB data are presented in [Bapat, 1993; Benz and Leischner, 1993]. The first paper proposes the use of *virtual attributes*, which are dynamically computed from other attributes to enforce access control. The second paper hints at future extensions of their modeling technique which would include views to filter certain objects. The role and applicability of formal model-based techniques for diagnosis of dynamic systems is analyzed in [Riese, 1993].

The (suspended) draft SNMPv2 framework defines "MIB views" as subsets of the managed objects held by the managed entity. These subsets are defined by inclusion and exclusion of subtree families via bit-masks, providing a simple projection of a subset of the MIB tree. The purpose of these "MIB views" is to define a *context*, that is, a collection of managed objects that can be accessed locally by an SNMPv2 entity. SNMPv2 provides a "context" mechanism to support a projection view of an MIB. A party may be authorized to access a subset of the MIB. MIB Views extend this mechanism to support not only projections but also computations over MIB data.

An alternative VDL, which also requires M_bD support was proposed by [Arai and Yemini, 1995]. Their VDL extends the MIB SMI to support views. The SMI is a subset

```
viewAtmIfTable VIEW-TYPE
    SYNTAX SEQUENCE OF
              vAtmIfTableEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION "ATM Interface Table"
    INDEX {vIfIndex}
    ::= {view 1}

vAtmIfTableEntry VIEW-TYPE
    SYNTAX VAtmIfTableEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION "Conceptual row"
    ::= {viewAtmIfTable 1}

VAtmIfTableEntry ::= SEQUENCE {
    vIfIndex INTEGER,
    vIfSpeed Gauge,
    vMaxVpcs INTEGER,
    vMaxVccs INTEGER
    }

vIfIndex VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION "ifIndex from ifTable"
    COMPUTED-BY func_vIfIndex
    ::= {vAtmIfTableEntry 1}

func_vIfIndex VIEW-FUNCTION
    SELECT ifIndex[SELF-INDEX]
    WHERE  ifType[SELF-INDEX] = 37

vIfSpeed VIEW-TYPE
    SYNTAX Gauge
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION "Interface speed"
    COMPUTED-BY func_vIfSpeed
    ::= {vAtmIfTableEntry 2}

func_vIfSpeed VIEW-FUNCTION
    SELECT ifSpeed[SELF-INDEX]
    WHERE  ifType[SELF-INDEX] = 37

vMaxVpcs VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-write
    STATUS current
    DESCRIPTION "Max. number of VPCs."
    COMPUTED-BY func_vMaxVpcs
    ::= {vAtmIfTableEntry 3}

func_vMaxVpcs VIEW-FUNCTION
    SELECT atmInterfaceMaxVpcs[SELF-INDEX]
    WHERE  ifType[SELF-INDEX] = 37

vMaxVccs VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-write
    STATUS current
    COMPUTED-BY func_vMaxVccs
    ::= {vAtmIfTableEntry 4}

func_vMaxVccs VIEW-FUNCTION
    SELECT atmInterfaceMaxVccs[SELF-INDEX]
    WHERE  ifType[SELF-INDEX] = 37
```

Figure 5.19: Arai's example of VDL statement to join tables.

of ASN.1, a data definition language used to define the types and representations of the contents of SNMP PDUs. Except for comments written in English, SMI does not have facilities to precisely define how values are computed. Extending the SMI to support views requires merging the SQL select-from-where constructs in the ASN.1 notation. This results in very long and detailed specifications of MIB views.

Consider, for instance, the simple example given in Figure 5.10, which only takes five lines in our VDL. The same example is given in Figure 5.19 using SMI extensions following the notation presented in [Arai and Yemini, 1995]. Notice also that the SMI is a well established standard at the core of SNMP. An extension or revision of such standard must also overcome many non-technical obstacles, as the experience of SNMPv2 has shown. Our approach, in contrast, does not require any changes to the SMI. The VDL translator will read the view definitions and generate correct MIB definitions to be compiled by any standard MIB compiler.

5.6 Conclusions

The lack of an appropriate external data model is one of the reasons for the dearth of applications that provide more functionality than simple browsing of MIB values. This chapter described the design of an MIB Computations System (MCS) that supports the dynamic definition of external data models for MIBs. The MCS consists

of (1) a *View Definition Language* to specify MIB external views and (2) SNMP-agent extensions that implement them. Collections of VDL views define a new *Virtual* MIB, V-MIB, that can be queried via SNMP.

The V-MIB variables are computed via instrumentations over an MIB, performed by a special M_bD-server. V-MIB tables which contain correlated data, to generate atomic snapshots of MIB data, establish access control mechanisms, select data which meet a filtering condition, and execute atomic actions. This is all achieved by a simple set of constructs which a management engineer can use without being proficient on the SMI. All the MIB structures and their semantic routines are automatically generated by the VDL translator, and executed by the MCVA. These computations can be accessed via SNMP.

6

Summary and Conclusions

This chapter summarizes the work presented and reviews the main contributions of this dissertation. The principal theme of this dissertation is to apply elastic processing technologies to support distributed system management.

Elastic Processing

The traditional Client/Server interaction model involves transfer of data and/or commands among *statically located* processes. Data and commands form the *mobile* parts of a computation while the processes are static. This model enforces a rigid association of functionality with server processes that often leads to a concentration and centralization of application logic. Such centralization results in inefficient and intrinsically unreliable distributed systems. There is a growing number of network computing scenarios which cannot be effectively addressed by such static interaction paradigms.

As computing processors become faster and network bandwidth increases, network interaction delays become the most critical performance problem for distributed applications. Rigid Client/Server distributed computing does not scale well to environments where network delays are relatively long compared with their local computations. Elasticity provides an effective way to overcome these delays, by taking advantage of CPU cycles and network bandwidth to move computations closer to the resources that they need to access.

Elastic processes support remote delegation of language-independent agents, a new construct for interaction between distributed applications. Elastic processing consists of two components: (1) a Remote Delegation Service (RDS) to dispatch an agent to a remote elastic process, invoke it as a thread of the process, and control its execution; (2) An elastic process structure to support dynamic delegation, linking, and remote control of agents. Elasticity is useful for many types of applications. Common characteristics of such applications are that (1) they are long-lived, (2) they execute over distributed heterogeneous environments, (3) they must adapt to changes in the environment, (4) they have real-time constraints, and (5) they must execute over hosts with insufficient computing resources or over low bandwidth networks.

Management by Delegation

Network and system management present a broad range of technical challenges. Current network management systems pursue a platform-centered paradigm which is unscalable, performs poorly, exposes the semantic heterogeneity of devices, and reflects obsolete performance tradeoffs. Current standards define the observation and collection of MIB data independently of its use. Since the need for MIB data cannot be predicted, many observations are collected and stored but never used.

M$_b$D supports the dynamic distribution and automation of management responsibilities to networked devices. Thus it addresses many of the fundamental limitations of platform-centric management systems. M$_b$D provides a simple yet powerful scheme to dynamically compose management applications using delegated agents. Designers and vendors of network devices can provide predefined programs that encapsulate network management expertise. Network managers use these programs as delegated management agents to compose distributed multi-process applications which they can configure and control.

M$_b$D-server allows the devices to perform an open-ended set of management programs in close proximity to the managed devices. Management applications gain much faster response time to management events like faults. M$_b$D-server executes these functions independently of the delegator's execution, unless coordination is required. Each delegated agent can be designed and coded as part of a specific management scheme. This can be done much later than the device MIB design and deployment, and can be tailored to the specific requirements of each network installation. M$_b$D is an effective tool to balance the computational requirements of management applications and respond to the vicissitudes of their network environments. Management applications can use delegated agents to quickly react to ephemeral conditions like network load and configuration changes like device replacements.

Compressing Management Data

We introduced a formal notation to describe the behaviors of managed entities and their observations by MIBs and by management applications. Management applications define composite observation operators to reach management decisions. Current network management paradigms introduce many problems that prevent the effective computation of such operators. They do not support the temporal distribution and spatial decentralization required to compute real-time management functions effectively. The dissertation introduced a framework for compressing real-time data and making management decisions via delegated *health* functions. The definition of what constitutes a *healthy* network cannot be standardized or fixed for all networks, since it is installation and time dependent. M$_b$D permits applications to configure observation processes to flexibly monitor information of their interests.

We defined index functions to compress real-time management observations at the location of the managed entities. We implemented and demonstrated an M$_b$D application that computes a health index of a network. This application performs

proactive diagnoses of failures and performs corrective actions in real-time. Management decisions, such as to temporarily disconnect a device, are executed efficiently without the need for remote NOC intervention. Real-time operational data is effectively compressed at the M_bD-server, reducing the management data overhead on the network. The M_bD environment allowed network managers to tailor and customize the health application during execution.

MIB Views

Management applications need to compute useful information from raw data collected in MIBs. The lack of an external-level data model repository results in excessive and redundant retrievals and recomputations. It is also one of the reasons for the dearth of management applications that provide more functionality than simple browsing of MIB values.

MIB views provide a mechanism to perform such computations at the devices while using standard data access protocols (e.g., SNMP) to access the results of the computations. We designed an M_bD-server that provides SNMP managed elements a framework generate external-level management information. Management engineers can use the VDL language to specify MIB views, which are implemented by the M_bD-server. The VDL language allows management engineers to create new, "virtual" MIB tables which contain correlated data, to generate atomic snapshots of MIB data, establish access control mechanisms, select data which meet a filtering condition, and execute atomic actions. These MIB extensions implement efficient computations that can deliver the required information to management applications via SNMP.

Bibliography

[Agrawal and Ezzat, 1987] Rakesh Agrawal and Ahmed K. Ezzat. Location Independent Remote Execution in NEST. *IEEE Transactions on Software Engineering*, 13(8):905–912, August 1987.

[Anderson, 1980] James P. Anderson. *Computer Security Threat Monitoring and Surveillance*. James P. Anderson Co., Fort Washington, PA, 1980.

[ANSI, 1989] ANSI. *Programming Language – C*. American National Standards Institute, 1989. ANSI/X3.159.

[Arai and Yemini, 1995] Kazushige Arai and Yechiam Yemini. View Definition Language (VDL) for SNMP. In *The Fourth International Symposium on Integrated Network Management*, pages 454–465, Santa Barbara, California, April 1995.

[Auerbach *et al.*, 1994a] Joshua Auerbach, Arthur Goldberg, Germán Goldszmidt, Ajei Gopal, Mark Kennedy, Josyula Rao, and James Russell. Concert/C: A Language for Distributed Programming. In *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, California, January 1994.

[Auerbach *et al.*, 1994b] Joshua Auerbach, Arthur Goldberg, Germán Goldszmidt, Ajei Gopal, Mark Kennedy, and James Russell. Concert/C Tutorial and User Guide: A Programmer's Guide to a Language for Distributed C Programming. Technical report, IBM T. J. Watson Research Center, 1994. ftp://software.watson.ibm.com/pub/concert/doc/userguide.ps.

[Autrata, 1991] Matthias Autrata. OSF Distributed Management Environment. In *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Santa Barbara, CA, October 1991.

[Bapat, 1993] Subodh Bapat. Richer Modeling Semantics for Management Information. In *The Third International Symposium on Integrated Network Management*, pages 15–28, San Francisco, CA, April 1993.

[Bauer *et al.*, 1993] Michael Bauer, Pat Finnigan, James Hong, Jan Pachl, and Toby Teorey. An Integrated Distributed System Management Architecture. In *Proceedings of the 1993 IBM Centre for Advanced Studies Conference*, pages 27–40, Toronto, Canada, October 1993.

[Ben-Artzi *et al.*, 1990] A. Ben-Artzi, A. Chadna, and U. Warrier. Network Management of TCP/IP Networks: Present and Future. *IEEE Network Magazine*, July 1990.

[Benz and Leischner, 1993] Ch. Benz and M. Leischner. A High Level Specification Technique for Modeling Networks and their Environments including Semantic Aspects. In *The Third International Symposium on Integrated Network Management*, pages 29–43, San Francisco, CA, April 1993.

[Bernstein and Yuhas, 1995] Lawrence Bernstein and C.M. Yuhas. Can we talk? In *The Fourth International Symposium on Integrated Network Management*, pages 670–676, Santa Barbara, California, April 1995.

[Bernstein, 1993] Lawrence Bernstein. The Vision for Networks and their Management, September 1993. An address at the IEEE Second Workshop on Network Management and Control.

[Birrell and Nelson, 1984] Andrew D. Birrell and Bruce J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[Borenstein, 1994] Nathaniel S. Borenstein. Email With a Mind Of Its Own: The Safe-Tcl Language for Enabled Mail, 1994. ftp://ftp.fv.com/pub/code/other/safe-tcl.tar.

[Braden and Schon, 1991] Robert Braden and Annette De Schon. *NNStat Internet Statistics Collection Package Introduction and User Guide*. USC Information Sciences Institute, January 1991.

[Califano and Rigoutsos, 1993] Andreas Califano and Isidore Rigoutsos. FLASH: A Fast Look-Up Algorithm for String Homology. In *Proceedings First International Conference on Intelligent Systems for Molecular Biology*, Washington, DC, July 1993.

[Carl-Mitchell and Quarterman, 1994] Smoot Carl-Mitchell and John S. Quarterman. How Wide Is The Internet. *RS/Magazine*, 3(9):24–27, September 1994.

[Case *et al.*, 1990] Jeffrey D. Case, Mark S. Fedor, Martin L. Schoffstall, and James R. Davin. A Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[Case *et al.*, 1993] Jeffrey D. Case, Keith McCloghrie, Marshall T. Rose, and Steven L. Waldbusser. Introduction to version 2 of the Internet-standard Network Management Framework. RFC 1441, April 1993.

[Cheriton, 1988] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[Chess *et al.*, 1995] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant Agents for Mobile Computing. Technical Report 20010, IBM Research, March 1995.

[Chu, 1993] Wesley W. Chu. System Management Research via Behavior Characterization. In *IEEE First International Workshop on Systems Management*, pages 1–6, Los Angeles, California, April 1993. IEEE Computer Society Press.

[Cohen and Feigenbaum, 1981] Paul R. Cohen and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume 3. William Kaufmann, Inc., Los Altos, California, 1981.

[Comer and Stevens, 1991] Douglas Comer and David Stevens. *Internetworking with TCP/IP, Volume II, Design, Implementation and Internals.* Prentice Hall, 1991.

[Comer and Stevens, 1993] Douglas Comer and David Stevens. *Internetworking with TCP/IP, Volume III, Client - Server Programming and Applications.* Prentice Hall, 1993.

[Comer, 1991] Douglas Comer. *Internetworking with TCP/IP, Principles, Protocols and Architecture.* Prentice Hall, 1991.

[DARPA, 1988] DARPA. *Neural Network Study.* AFCEA International Press, Fairfax, Virginia, November 1988.

[Duda and Hart, 1973] R. O. Duda and P.E. Hart. *Pattern Classification And Scene Analysis.* John-Wiley & Sons, New York, 1973.

[Dupuy *et al.*, 1989] Alex Dupuy, Jed Schwartz, Yechiam Yemini, Gil Barzilai, and Albert Cahana. Network Fault Management A User's View. In Branislav N. Meandzija and Jil Westcott, editors, *The First IFIP International Symposium on Integrated Network Management*, pages 101–107, Boston, MA, May 1989. North Holland.

[Dupuy, 1995] Alex Dupuy. *Smarts Operations Server.* Smarts, 1995.

[Eckerson, 1992] W. Eckerson. Net Management Traffic Can Sap Net Performance. *Network World*, May 1992.

[Elmasri and Navathe, 1989] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, 1989.

[Erlinger *et al.*, 1994] Mike Erlinger, Elicia Engelman, Nathan Tuck, Adam Wells, Christopher White, and Philip Winston. Demonstrating High Performance Scalable Network Management Paradigms. Technical report, Harvey Mudd College, December 16 1994.

[Erlinger, 1993] Michael Erlinger. RMON - From Concept to Specification. In Heinz-Gerd Hegering and Yechiam Yemini, editors, *The Third IFIP International Symposium on Integrated Network Management*, pages 73–80, San Francisco, CA, April 1993.

[Falcone, 1987] Joseph R. Falcone. A Programmable Interface Language for Heterogenous Distributed Systems. *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.

[Goldszmidt and Yemini, 1991] Germán Goldszmidt and Yechiam Yemini. The Design of a Management Delegation Engine. In *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Santa Barbara, CA, October 1991.

[Goldszmidt and Yemini, 1993] Germán Goldszmidt and Yechiam Yemini. Evaluating Management Decisions via Delegation. In *The Third International Symposium on Integrated Network Management*, San Francisco, CA, April 1993.

[Goldszmidt and Yemini, 1995] Germán Goldszmidt and Yechiam Yemini. Distributed Management by Delegation. In *The 15th International Conference on Distributed Computing Systems*. IEEE Computer Society, June 1995.

[Goldszmidt *et al.*, 1990] Germán Goldszmidt, Shmuel Katz, and Shaula Yemini. High Level Language Debugging for Concurrent Programs. *ACM Transactions on Computer Systems*, 8(4):311–336, November 1990.

[Goldszmidt *et al.*, 1991] Germán Goldszmidt, Yechiam Yemini, and Shaula Yemini. Network Management: The MAD Approach. In *Proceedings of the IBM/CAS Conference*, Toronto, Canada, October 1991.

[Goldszmidt, 1992] Germán Goldszmidt. Elastic Servers in Cords. In *Proceedings of the 2nd IBM/CAS Conference*, Toronto, Canada, November 1992.

[Goldszmidt, 1993a] Germán Goldszmidt. Distributed System Management via Elastic Servers. In *IEEE First International Workshop on Systems Management*, pages 31–35, Los Angeles, California, April 1993.

[Goldszmidt, 1993b] Germán Goldszmidt. On Distributed System Management. In *Proceedings of the Third IBM/CAS Conference*, Toronto, Canada, October 1993.

[Gosling and McGilton, 1995] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, 1995.

[Gosling, 1986] James Gosling. Sundew: A distributed and extensible window system. In *Proceedings of the 1986 Winter Usenix Technical Conference*, pages 98–103, Boulder, CO, January 1986.

[Gosling, 1995] James Gosling. *The HotJava Browser: A White Paper*. Sun, 1995.

[Harrison *et al.*, 1995] Colin Harrison, David Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? Technical Report RC 19887, IBM Research, March 1995.

[Hedrick, 1988] C Hedrick. Routing Information Protocol. RFC 1058, June 1988.

[Hennessy and Patterson, 1990] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[ISO, 1989] International Standards Organization - ISO. Information Processing - Open System Interconnection - Basic Reference Model Part 4 - OSI Management Framework. New York, USA, January 18 1989.

[ISO, 1990a] International Standards Organization ISO. Open Systems Interconnection - Common Management Information Protocol Specification, 1990. International Standard 9596.

[ISO, 1990b] International Standards Organization ISO. Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation ONE (ASN.1), 1990. International Standard 8825.

[Jacobson, 1988] Van Jacobson. Traceroute Software, December 1988. ftp://ftp.ee.lbl.gov/pub/traceroute.tar.Z.

[Leinwand and Fang, 1993] Allan Leinwand and Karen Fang. *Network Management A Practical Perspective*. Addison-Wesley Publishing Company, 1993.

[Lewine, 1991] Donald Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, Inc., 1991.

[Liskov and Shrira, 1988] Barbara Liskov and Liuba Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *ACM SIGPLAN*, pages 22–24, Atlanta, GA, June 1988.

[M.Ahmed and K.Tesink, 1994] M.Ahmed and K.Tesink. Definitions of Managed Objects for ATM Management, Version 8.0. RFC 1695, August 1994. Bell Communications Research.

[Markov, 1993] John Markov. Microsoft Soft-pedaling its Latest, May 24 1993. The New York Times.

[McCloghrie and Rose, 1991] K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets: MIB-II. RFC 1213, March 1991.

[Meandzija *et al.*, 1991] B.N. Meandzija, K.W. Kappel, and P.J. Brusil. Integrated Network Management and The International Symposia. In Iyengar Krishnan and Wolfgang Zimmer, editors, *The Second International Symposium on Integrated Network Management*, Washington, DC, April 1991. North Holland.

[Meleis *et al.*, 1994] Hanafy Meleis, David Su, Wayne McCoy, and Michael Nowak. Proposed Focused Program on: Operations and Management of Information Networks, October 1994. NIST - Advanced Technology Program.

[Meyer *et al.*, 1993] Kraig Meyer, Joe Betser, Eric Negaard, Dennis Persinger, Steven Wang, Robert Maltese, and Carl Sunshine. An Architecture Driven Comparison of Network Management Systems. In Heinz-Gerd Hegering and Yechiam Yemini, editors, *The Third IFIP International Symposium on Integrated Network Management*, pages 479–492, San Francisco, CA, April 1993.

[Meyer *et al.*, 1995] Kraig Meyer, Mike Erlinger, Joe Betser, Carl Sunshine, Germán Goldszmidt, and Yechiam Yemini. Decentralizing Control and Intelligence in Network Management. In *The Fourth International Symposium on Integrated Network Management*, May 1995.

[Ousterhout, 1990] John K. Ousterhout. Tcl: An Embeddable Command Language. In *Proceedings of the 1990 Winter USENIX Conference*, 1990.

[Partridge, 1992] Craig Partridge. *Late-Binding RPC: A Paradigm for Distributed Computation in a Gigabit Environment.* PhD thesis, Harvard University, March 1992.

[Porras, 1992] Philip Andrew Porras. Stat a state transition analysis tool for intrusion detection. Master's thesis, UC Santa Barbara, July 1992. TRCS93-25.

[Rahali and Gaiti, 1991] Ilham Rahali and Dominique Gaiti. A multi-agent system for network management. In *The Second International Symposium on Integrated Network Management*, pages 469–479, Washington, DC, April 1991.

[Rich and Knight, 1991] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw Hill, 2nd edition, 1991.

[Riecken, 1994a] Doug Riecken. A Conversation with Marvin Minsky About Agents. *Communications of the ACM*, 37(7):23–29, July 1994.

[Riecken, 1994b] Doug Riecken. Intelligent Agents. *Communications of the ACM*, 37(7):18–21, July 1994.

[Riese, 1993] Marc Riese. *Model-Based Diagnosis of Communication Protocols.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1993. Available via anonymous ftp from litsun.epfl.ch.

[Rivest, 1992] Ron Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

[Rose, 1991a] Marshall T. Rose. Network Management is Simple: you just need the "right" framework. In *The Second International Symposium on Integrated Network Management*, pages 9–23, Washington, DC, April 1991.

[Rose, 1991b] Marshall T. Rose. *The Simple Book, An introduction to Management of TCP/IP-based Internets.* Prentice Hall, 1991.

[Rose, 1994] Marshall T. Rose. *The Simple Book, An Introduction to Internet Management.* Prentice Hall, 2nd edition, 1994.

[Simpson, 1993a] William Simpson. PPP in HDLC Framing. RFC 1549, December 1993.

[Simpson, 1993b] William Simpson. The Point-to-Point Protocol (PPP). RFC 1548, December 1993.

[Sloman, 1994] Morris Sloman, editor. *Network and Distributed System Management.* Addison-Wesley Publishing Company, 1994.

[Smith, 1988] Jonathan M. Smith. A Survey of Process Migration Mechanisms. *Operating Systems Review*, 22(3):28–40, 1988.

[Soares and Karben, 1993] Patricia Gomes Soares and Alan Randolph Karben. Implementing a Delegation Model Design of an HPCC Application Using Concert/C. In *Proceedings of the 1993 IBM Centre for Advanced Studies Conference*, pages 729–738, Toronto, Canada, October 1993.

[Soares, 1992] Patricia Gomes Soares. On Remote Procedure Call. In *Proceedings of the Second CASCON International Conference*, Toronto, Canada, November 1992.

[Sollins, 1992] Karen R. Sollins. The TFTP Protocol (Revision 2). RFC 1350, July 1992.

[Stallings, 1993] William Stallings. *SNMP, SNMPv2, and CMIP The Practical Guide to Network Management Standards.* Addison-Wesley Publishing Company, 1993.

[Stamos and Gifford, 1990a] James W. Stamos and David K. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.

[Stamos and Gifford, 1990b] James W. Stamos and David K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[Stevens, 1990] W. Richard Stevens. *Unix Network Programming*. Prentice Hall Software Series, 1990.

[Stine, 1990] Robert H. Stine. FYI on a Network Management Tool Catalog: Tools for Monitoring and Debugging TCP/IP Internets and Interconnected Devices. RFC 1147, April 1990.

[Synoptics, 1990] Synoptics. LattisNet SNMP Ethernet Concentrator MIB for TCP/IP Networks., 1990. Private MIB.

[Tanembaum and van Renesse, 1988] Andrew S. Tanembaum and Robert van Renesse. A critique of the Remote Procedure Call Paradigm. *Research into Networks and Distributed Applications*, pages 775–783, April 1988.

[Tanenbaum, 1992] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[Terplan, 1987] Kornel Terplan. *Communication Networks Management*. Prentice Hall, 1987.

[Tesink and Brunner, 1994] Kaj Tesink and Ted Brunner. (Re)Configuration of ATM Virtual Connections with SNMP. *The Simple Times*, 3(2), August 1994.

[Theimer and Hayes, 1991] Marvin M. Theimer and Barry Hayes. Heterogeneous Process Migration by Recompilation. In *The 11th International Conference on Distributed Computing Systems*, pages 18–25, Arlington, Texas, May 1991. IEEE.

[Uni, 1986] University of California, Berkeley. *Unix System Manager's Manual (SMM)*, April 1986.

[White, 1994] James E. White. Telescript Technology: The Foundation for the Electronic Marketplace, 1994. General Magic White Paper.

[Yemini *et al.*, 1989] Shaula Yemini, Germán Goldszmidt, Alex Stoyenko, Yi-Hsiu Wei, and Langdon Beeck. Concert: A High-Level-Language Approach to Heterogeneous Distributed Systems. In *The Ninth International Conference on Distributed Computing Systems*, pages 162–171. IEEE Computer Society, June 1989.

[Yemini *et al.*, 1991] Yechiam Yemini, Germán Goldszmidt, and Shaula Yemini. Network Management by Delegation. In *The Second International Symposium on Integrated Network Management*, pages 95–107, Washington, DC, April 1991.

[Yemini, 1993] Yechiam Yemini. The OSI Network Management Model. *IEEE Communications Magazine*, pages 20–29, May 1993.

[Zimmerman, 1991] D. Zimmerman. The Finger User Information Protocol. RFC 1288, December 1991. Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University.

Appendix A

Network Management Standards and Models

A.1 Background

Most organizations depend on networked systems to support their critical information functions. Hence they are exposed to the inherent risks associated with operating such systems. For example, their information systems are vulnerable to network failures and security compromises. These problems can have severe costs for many types of businesses, and in particular financial institutions. From an organization's perspective, failures have many different manifestations. As Arno Penzias states it, *"If a customer can't use it, it might as well be broken– it might as well not exist!"* [Bernstein and Yuhas, 1995]. Network failures can result in significant physical harm to people. For example, failures in the London Ambulance Service dispatching network resulted in almost twenty deaths as a result of ambulance delays of up to three hours [Bernstein, 1993]. As this example shows, an effective management system that ensures fault-free, efficient and secure operations is, literally, a vital need.

Modern computing technologies bring huge benefits to society. Realizing these benefits requires more than simply deploying new hardware and software products. Because of our dependency on the proper operations of information systems, we must ensure that these assets are properly managed. Failure to allocate sufficient resources for network and system management will limit the benefits and raise the costs of these newer technologies. An effective management framework that enables the proper utilization of these resources is, therefore, of paramount importance.

The main goal of network management is to maintain the communication network operating efficiently at all times. Network administrators and operators that manage large distributed systems need automated tools to maintain seemly operating networks. An integrated management environment must, therefore, enable network operators to make timely management decisions. To be effective, network managers must first overcome the volume and complexity of management information that characterize large heterogeneous distributed systems. To achieve this, they must use automated management tools to filter and sort the management information that they need from a sea of management data.

Several emerging trends contribute to make networked systems management complex and technically challenging. First, the explosive growth of the number of devices that are attached to networks. Second, the heterogeneity of these devices, in terms of brands and types. Third, the increased functionality and complexity of newer devices. Finally, the fact that organizations are moving away from centralized, mainframe based *"glass-house"* environments and towards departmental, distributed computing environments.

Distributed system management presents a broad range of non-trivial technical challenges, and their research is at an early stage. Many of the fundamental technical problems involved in accomplishing manageability are not fully understood. Indeed, "It is not clear that we know what data should be collected, how to analyze it when we get it, or how to structure our collection systems" [Stine, 1990]. For example, typical diagnostic problems arising in large-scale heterogeneous systems include [Dupuy *et al.*, 1989]:

- Faults that are unobservable, either because of systems deadlocks or because of lack of device instrumentation.
- Partial observations that provide insufficient data to pinpoint a problem.
- Uncertainty in the observations, sometimes due to multiple potential causes.
- Too many related observations obscuring the real problem.
- Local recovery procedures that may destroy evidence.

Many management functions presently require manual monitoring and reactive analysis by expert operations personnel. Examples of such tasks include inventory control of installed equipment and software, discovery of intrusion attempts, and handling of device breakdowns. A labor-intensive management paradigm exposes enterprises to many operations failures. It has also caused management costs to consume an increasing and dominant share[1] of information systems budgets. For example, a major software developer reported that maintenance and operations account for as much as 90% of the cost of business computing [Markov, 1993]. The total costs associated with the operation and management of information networks for the USA economy in 1994 has been estimated at over $200 billion dollars [Meleis *et al.*, 1994].

Networks were not designed to be managed

Unfortunately, most networks and devices were not designed with integrated management facilities. In the early days of the Internet, for instance, management schemes were based on personal cooperation among the researchers who built and used the Internet [Comer, 1991]. Before 1990, there were few generic network management tools available, and they provided very basic functions. For instance, operators used `traceroute` [Jacobson, 1988] to track the route that IP packets follow or to find a "miscreant" gateway that is discarding some packets. Some private-vendor networks supported more sophisticated but proprietary management tools, e.g., IBM's NETVIEW for SNA networks. An overview of such systems is presented in [Terplan, 1987].

[1] The Gartner Group estimates are 65-90%.

As private and public networks grew in size and complexity, more formal methodologies and automated methods became necessary. Most useful management tools were vendor-specific and supported only a certain class of devices. Network operators became unable to handle non-amenable problems in real-time, for an ever growing melange of devices. This situation created a demand for *integrated* management solutions that are vendor-neutral and can support any device. Standards organizations ameliorated this situation by providing network management frameworks such as the IETF's SNMP [Case *et al.*, 1990], and the ISO's CMIP [ISO, 1990a].

A.2 Network and System Management Functions

Network managers need assistance for (a) controlling and securing assets connected to the network, and (b) improving the overall quality of information services. Managers must control who can access network resources, and provide satisfactory service to their users. To achieve these high-level goals, management systems must implement specific management functions. These management functions are often divided into five major areas [ISO, 1989]: fault, accounting, configuration, performance, and security.

1. *Fault management* functions detect and correct abnormal behaviors. For example, they handle device breakdowns (e.g., file server failures) and network cleavages (e.g., connectivity lost due to repeater or cable failures).

2. *Accounting management* functions collect and process resource consumption data. For instance, they can compute the billing cost of using a videoconferencing application.

3. *Configuration management* functions detect and control the state of the network resources. For instance, they can track the use of shared software licenses by different users.

4. *Performance management* functions evaluate the effectiveness of communications. For instance, they monitor response time delays and uneven packet traffic distributions.

5. *Security management* functions monitor and control access to resources, e.g., unauthorized reading of database records.

The development of management protocol standards has served as the key driving force of the network management field since the late 1980s. The main standards are the IETF's *Simple Network Management Protocol*[2], SNMP [Case *et al.*, 1990], and the OSI's *Common Management Information Protocol*, CMIP [ISO, 1990a]. Both standards follow a platform-centered paradigm, and provide a database model to access device data and instrumentation. The original goal of these standard efforts was to enable the implementation of enterprise-wide management across heterogeneous devices. As described in more detail in the following sections, the platform-centered paradigm subsumed by the standards has serious limitations.

Standardization efforts in both OSI and the IETF have focused primarily on the management protocol and the structure of management information. A comparison of

[2]A new draft version, SNMPv2 [Case *et al.*, 1993], has been under development for some time, but its working group was suspended on September 1995.

several network management system products based on these standards is given in [Meyer *et al.*, 1993]. The following sections briefly outline SNMP and CMIP. We provide a more detailed presentation and critical analysis of SNMP, since we will use SNMP for illustration purposes as a representative protocol. Our results, however, are equally valid for the OSI management framework and its protocol CMIP. Indeed, their limitations result from their common management paradigm rather than from specific design features of each standard.

Different administrative domains have different requirements and need different management strategies [Chu, 1993]. Thus, they may require different technologies to organize and access their management information efficiently. Given the existing plethora of manageable resources, the hodgepodge of administrative approaches, inconsistent tools, and inadequate facilities [Autrata, 1991], managing a distributed system is a difficult task. For example, some installations may need expert system databases and technologies (e.g., [Rahali and Gaiti, 1991]). Other organizations may use model-based diagnosis techniques (e.g., [Riese, 1993]), and so forth.

The network management community has been seeking a single unifying scheme, defined by a standard protocol and MIBs. Such an approach is, obviously, not a panacea for all the difficulties of managing a distributed system. Thus, it is preferable to develop a framework that admits many different management protocols and MIB types. SNMP in particular, is inadequate for managing complex devices. Still, SNMP is so ubiquitous and pervasive that it will continue to play a critical role for a long time. Similarly, the telecommunications industry will continue to use CMIP for their own reasons. Thus, there is a need for a management paradigm that can admit and support several management protocols concurrently.

A.3 Critical Evaluation of SNMP

To increase the ubiquity of network management agents, a minimalist approach "axiom" has driven the design of SNMP: *"The impact of adding network management to managed nodes must be minimal, reflecting a lowest common denominator"* [Rose, 1991a]. The design of SNMP is indeed simple, making it inexpensive for device vendors to implement[3]. This simplicity resulted in its rapid deployment across a wide variety of devices. Thus it became the primary method to retrieve management data from just about any device attached to an IP network. The SNMP paradigm has been successful for two primary reasons. First, most of the early network management applications followed a centralized paradigm which required a more global, or distributed knowledge that is not available locally. Second, the networked devices where agents need to execute did not have the sufficient computing resources to manage themselves.

Adherence to the above axiom resulted in network management systems and applications that are platform-centric, and do not provide all the capabilities needed to properly manage a network. For instance, later sections show that SNMP does not provide appropriate mechanisms to proactively respond to network failures in real-time. With the rapid

[3]Note, however, that the implementation of the SNMP protocol is large compared with other protocols of the TCP/IP suite. An analysis of implementations reports that *"SNMP accounts for nearly as much code [24.1%] as TCP [25.9%] even though it provides only a simple service."* [Comer and Stevens, 1991].

growth in the size of networks, scalability and performance became a growing concern. Presently and in the future, the embedded resources that can be allocated to management far exceed those assumed by the SNMP model. For example, network hubs come along with RISC microprocessors. Given such a powerful embedded processor committed to support management, it is wasteful to limit its use just to move data, when it can also be used to process applications. Devices that are capable of performing sophisticated management functions locally can take computing pressure off the centralized platforms, and reduce the network overhead of management messages. M_bD allows management applications to take advantage of these resources.

SNMP deficiencies

SNMP has many deficiencies. For example, SNMP polling introduces significant delays in retrieving management data to the platform. These delays are due to: (1) transient conditions, e.g., network contention or congestion, (2) configuration problems, e.g., the routing distance between the devices and the platform, and (3) the protocol design, e.g., the need for ASN.1 parsing of management PDUs in both communication endpoints (device and platform). High frequency polling introduces large bandwidth overhead. Slow polling will miss transient spikes (errors, load, etc) as it will average it over long periods of time.

SNMP-agent implementations introduce big timing errors in the observations of real devices, which produce outdated, and potentially erroneous, data in the agent's MIBs. Typically, MIB tables change while a management application is retrieving or examining them. Inaccuracies like these often lead to erroneous computations. To some extent, network operators can not trust the complete accuracy of their measurement instruments.

The following list outlines several of the problems associated with SNMP implementations (see [Ben-Artzi *et al.*, 1990] for a more detailed analysis). Our thesis is that M_bD provides mechanisms that address these problems.

- SNMP uses the network to transmit information about network measurements. Thus, it introduces an intrinsic disturbance i.e., a "probe" or Heisenberg effect.
- When a device is loaded, its SNMP-agent is scheduled with relatively lower priority, and thus queries to it will often be delayed.
- There is a large amount of relevant data for management purposes which is not available in any SNMP MIB. For instance, relevant data can be obtained via local instrumentation tools like `netstat`.
- SNMP is not well suited for retrieving large volumes of data, particularly large tables; We describe this problem in detail in Chapter 5.
- Event report traps are unacknowledged and an unreliable protocol (UDP) is used to deliver them. Thus, an agent cannot be sure that a trap has reached its destination;
- SNMP provides only trivial authentication via community strings, which makes it very vulnerable to attacks[4];

[4]Because of this, many implementations do not support the `Set` operator, limiting SNMP to monitoring only.

- SNMP does not directly support imperative commands with parameters; Section 5.4 describes this problem in detail.
- The MIB model does not support queries based on object values or types. Thus, applications can not filter MIB data at its source, and must retrieve large amounts of MIB data. This problem is elaborated on Section 5.3.1.
- Many implementations of SNMP-agents are erroneous and return wrong data. Ben Artzi describes controlled experiments which demonstrate significant inaccuracies in several commercial SNMP-agents [Ben-Artzi *et al.*, 1990].

A.4 Computations on MIBs

Large networks have many types of devices with different MIBs. The semantic heterogeneity of these MIBs complicates the development of management applications. Standard management protocols, like SNMP, unify only the *syntax* of managed data, not their *semantics*. For example, an MIB variable can be defined as an integer counter, which is documented as counting ethernet frames. However, the actual definition of how the counter observes frames is left for the implementor of the agent. Management software can do little with this data in the absence of a uniform semantic model for its interpretation. MIBs allow substantial semantic variations and differences in the implementation-specific behaviors of similar devices. The method that implements each MIB variable can be implemented in substantially different ways. For instance, routers from different vendors are often so different that manager applications *must* use vendor-specific private MIBs to handle them.

The manner in which devices are operated and controlled is often inseparable from their competitive advantages in the marketplace. Device vendors are interested in encouraging the growth of the diversity and flexibility of their devices, rather than subjecting them to a uniform "straight-jacket". Furthermore, because of the rapid pace of innovation in networking technologies, standard bodies are unable to make workable MIB definitions on time. Thus, vendors must resort to using different private MIBs. Private MIBs contain diverse and useful management features. While the data access mechanisms are fixed and standardized by the management protocols, the contents of the MIBs keep growing as new hardware is introduced. For example, Synoptics [Synoptics, 1990] defines private extensions in a separate MIB sub-tree under:
`iso(1).org(3).dod(6).internet(1).private(4).enterprises(1).synoptics(45).`
This subtree includes sections for each series of devices. The System 3000 Chassis, for instance, contains 87 objects. For example, `s3ChassisPsStatus` defines the chassis power supply status.

Much of the really relevant management data is found at that vendor-specific level. Indeed, it has been observed by many application designers that most *useful* information is specific to the implementation of a single device and describes its internals. For example, to find out why a device queue is overflowing it may be necessary to look into its buffer allocation scheme. Proprietary tools can make use of this private information. For instance, a management tool can correlate the actual semantics of some MIB variables with the non-standard methods to manipulate them.

Thus, a management application may change the number of buffers allocated to respond to a queue overflow.

Unfortunately, many MIBs are poorly documented. For instance, they typically do not describe how a given variable is calculated and how it is to be used. MIBs provide data, but they don't tell the management station what kind of problems can be solved with it. Furthermore, MIB specifications rarely assist the implementors of management applications on how to use this data in an intelligent way.

The semantic heterogeneity of managed data complicates the development of generic management software. In the absence of such software, platform-centered management applications are reduced to core-dumping cryptic device data on operators' screens, i.e., *"*MIB *browsers"*. Indeed, *"most network management systems are passive and offer little more than interfaces to raw or partly aggregated and/or correlated data in* MIB*s"* [Meandzija *et al.*, 1991]. MIB browsing is not an adequate model for managing networks, as there are very few adept operators able to decipher and interpret MIB contents. Typical end-users cannot interpret MIB data, and their organizations often cannot afford the costs of applications development and support. Given the wide variety of networked devices, and their large number of configuration options, there is an explosion of management information that needs to be handled. To understand and take advantage of the large amount of MIB data, it is sometimes necessary to bring together several experts, familiar with the various devices to identify a single problem.

Delegated agents may be designed to handle the specific operational environment and distinct features of specific resources. They require neither a uniform semantic model of device data nor adaptation to different platform environments. Thus, M$_b$D simplifies the problems of data heterogeneity that present barriers to the development of management applications. A device vendor can develop management application programs maintaining minimal components that can be easily ported. The exposure of these programs to heterogeneity can be handled via carefully defined minimal OCP interfaces. Contrast this with the task of developing platform applications (e.g., for fault management) that can handle the semantic heterogeneity of different devices.

A.5 CMIP

The OSI management standards define an object-oriented interaction model. The definition of a managed object class is specified by a template and consists of attributes, operations that can be applied to the object, behavior exhibited by the object in response to operations, and notifications that can be emitted by the object. The definition of a managed object is fixed at design time, i.e., it cannot change during execution. The exchange of management information between managers and agents is defined by the *CMI-Service* (CMIS), and its protocol (CMIP).

CMIS provides management-operation services that include `M-GET` to retrieve data, `M-SET` to modify data, `M-ACTION` to request the execution of an action, `M-CREATE`

(`M-DELETE`) to request the creation (deletion) of an instance of a managed object, and `M-CANCEL-GET` to cancel an outstanding `M-GET` request. Agents may report events about managed objects using `M-EVENT-REPORT`. In addition, CMIS provides *scoping, filtering, and synchronization* tools to select the objects which are subject to a management operation. Scoping is the identification of objects to which a filter is to be applied. Filtering consists of Boolean expressions about the presence or values of attributes in an object. If several objects are selected by a given scope and filter, two types of synchronization may be requested: *atomic* or *best-effort*.

The CMIP framework provides a much richer functionality than SNMP. However, because of the overall complexity and size of CMIP[5] many claim that this is *"a case of the cure's being worse than the disease"* [Stallings, 1993]. In practice, CMIP has received little attention outside the telecommunications industry. A detailed analysis of the OSI network management model is given in [Yemini, 1993].

[5] *"*CMIP *is really an abbreviation for the (overly) Complex Management Information Protocol"* [Rose, 1991b].

Appendix B

Glossary of Acronyms

API Application Programming Interface

A-RPC Asynchronous RPC

ASN.1 Abstract Syntax Notation 1

ATM Asynchronous Transfer Mode

CMIP Common Management Information Protocol

CMIS Common Management Information Service

DDL Data Definition Language

DP Delegated Program (agent code)

DPI Delegated Program Instance

DBM Delegation Backplane Middleware

FDC Full-Process DPI Controller

GHO Generic Health Object

IETF Internet Engineering Task Force

ILMI Interim Local Management Interface (for ATM)

IPC Inter Process Communication

MbD Management By Delegation

MD5 Message Digest 5

MIB Management Information Base

MCS MIB Computations System

MCVA MIB Computations of Views Agent

NMS Network Management Station

NOC Network Operating Center

OCP Observation and Control Point

PDA Personal Digital Assistant

PDU Protocol Data Unit

PVN Private Virtual Network

RDP Remote Delegation Protocol

RDS Remote Delegation Service

RPC Remote Procedure Call

RMON Remote Monitoring (an MIB)

SMI Structure of Management Information

SNMP Simple Network Management Protocol

SOS Smarts Operating Server

URL Universal Resource Locator

VDL View Definition Language

VMIB Virtual MIB