System Designs for Adaptive, Distributed Network Monitoring and Control

H. Li, S. Yang, H. Xi, and J. S. Baras
Center for Satellite and Hybrid Communication Networks
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD 20742
{hjli, syang, hxi, baras} @glue.umd.edu

Abstract

We present system designs for adaptive, distributed network monitoring and control. The ideas are to distribute some processing intelligence to network elements, and to design a dynamic interface such that the delegated agents could be managed remotely by the manager. The functionality of the delegated agents, and even that of the native software processes, could be extended dynamically without recompilation. The creation, deployment, operation, and management of the delegated agents require a standard infrastructure on each system where these agents need to be hosted. We use Java technology and C/C++ dynamic linkage mechanism to fulfill such a requirement under different situations and our system designs span a wide scope of applications.

Keywords

Network Management, Network Monitoring, Mobile Code, Java

1. Introduction

The increasing complexity and importance of communication networks have given rise to a steadily high demand for advanced network management. Network management system handles problems related to the configurability, reliability, efficiency, security and accountability of the managed distributed computing environments. Accurate and effective monitoring and control is fundamental and critical for all network management functional areas.

A conventional network management system consists of two classes of components: *managers* and *agents*. Applications in the management station assume the manager role; Agents are server processes running in each involved manageable network entity. These agents collect network device data, stores them in some format, and support a management protocol, e.g., Simple Network Management Protocol (SNMP) [13,14]. Manager applications retrieve data from element agents by sending corresponding requests over the management protocol.

0-7803-6719-7/01/$10.00 (c) 2001 IEEE

Such a system favors a centralized framework and works well for small networks. But as the networks become larger, the centralized paradigm will incur vast amounts of communication between manager and agent and thus occupy too much bandwidth unwisely. In this regard, we borrow the idea from [4] and *distribute* some processing logic by embedding the network elements with some codes for more responsibility. The embedded code within the network element is called a *delegated agent*. For theory and practice of agent technology, we refer to [3,16].

In telecommunications arena, the capability of placing new or added functionality into network elements is extremely important, especially as it potentially provides network operators with a mechanism for dynamically updating network elements processing logic. In conventional network monitoring systems, however, the set of services offered by the element agents is fixed and is accessible through interfaces that are statically defined and implemented. To this end, not only do we need to distribute intelligence, we also need to provide a dynamically extensible interface between such agents and the manager, such that the manager could change the parameter values, and extend the processing logic of the delegated agents dynamically.

Further, the functionality of the underlying native processes could also be dynamically extended via our *callback* mechanisms. The native processes, probably written in C/C++, embody the processing logic viewed as necessary at the time of native software developing. They may, however, lack of some considerations on some unanticipated cases. Such unanticipated cases, if they do occur, might lead to inconsistency in the processing followed. Thus we need to modify or extend the native software processing logic somehow to accommodate those unanticipated cases. Based on the observation that we would not like to re-code the C/C++ programs and recompile, reinstall, and reinstantiate the server processes, we need a flexible way such that the processing logic could be extended *dynamically*. Here, we respect the current processing logic and put on more processing capabilities to handle the unexpected cases. This is more like putting a "booster" rather than replacing the original logic..

To make it possible for agents to exist in heterogeneous environments, there needs to be a standard infrastructure on each system where they need to be hosted. Then agents may be developed as if they will be always on the same machine—the Virtual Machine, which could be but not limited to Java Virtual Machine (JVM). In this paper, we use either JVM or C/C++ dynamic linkage technology to serve as the Virtual Machine under different situations.

2. Related Work

Management by Delegation

Management by Delegation (MbD) [4] is one of the earliest efforts towards decentralization and increased flexibility of management functionality, and it greatly influenced later research and exploration along this direction [2,12]. However, there are two disadvantages with MbD that probably have affected its application: (1) The proof-of-concept MbD system was implemented with a proprietary server environment and we hardly see any working systems that are built upon this proprietary

environment. (2) The MbD server environment is so comprehensive and complicated that it can turn out to be an "overkill" in most real-world applications. Still, we must give credit to MbD because it can be considered a precursor of the ideas discussed in this paper. The major difference is that we have adopted the standard Java or C/C++ platform and, from the very beginning, aimed to build a portable, simple, yet powerful framework that can be easily understood, implemented and enhanced.

Mobile Agent

Using mobile agents in decentralized and intelligent network management [2,5,9,10] is a great leap from client-server based management pattern. The problem is that it proposed a change to network management paradigm that is so radical that even the authors themselves realized that [2] "further validation with quantitative data" would be necessary to prove its effectiveness. In contrast, our system exploits the idea of code-on-demand [3], still retains client-server architecture, and assumes a management server in each device concerned. Therefore, comparing with their mobile agent counterparts, the behaviors of our agents are much easier to understand and anticipate, making them more straightforward to integrate and co-exist with traditional systems.

Web-based Network Management

We are by no means the first people thinking of using Java technology in network management [6,11]. Web-based Network Management is a well-justified idea that attempts to provide uniform management services through such common client-side interface as Web browsers. Instead, we have used Java for a totally different purpose, which is not to facilitate client-side presentation or Web integration, but to use Java's native support for distributed computing, remote class downloading and object serialization to implement dynamic and intelligent network monitoring. However, it makes perfect sense to include Web-based front-ends into our systems.

Section 3 describes the conceptual structure of the system, followed by the system design considerations in section 4. In section 5 we give two classes of system designs that use Java or native code technology and present the trade-offs between them. We conclude the paper in section 6.

3. System Architecture

The system has been realized by a set of adaptable network element management agents and a network manager-coordinator, as shown in Figure 3-1.

The adaptable network element management agent provides the network Manager-Coordinator with an information view of the supported network element management information. Such an agent possesses a Collection API, by which the agent can dynamically change the way the information is collected from the network element.

The network manager-coordinator is responsible for accessing the network management information provided by the delegated agents, coordinating the dynamic definition of the information view of such management information, and coordinating the way the network management information is collected from the network elements

by the delegated agents. The network manager-coordinator coordinates the dynamic update of the information views by specifying the specific filtering expressions and the various threshold values. When a threshold crossing is detected an asynchronous notification will be forwarded to the manager-coordinator. This event-based paradigm for network monitoring results in huge reduction of monitoring traffic [1].

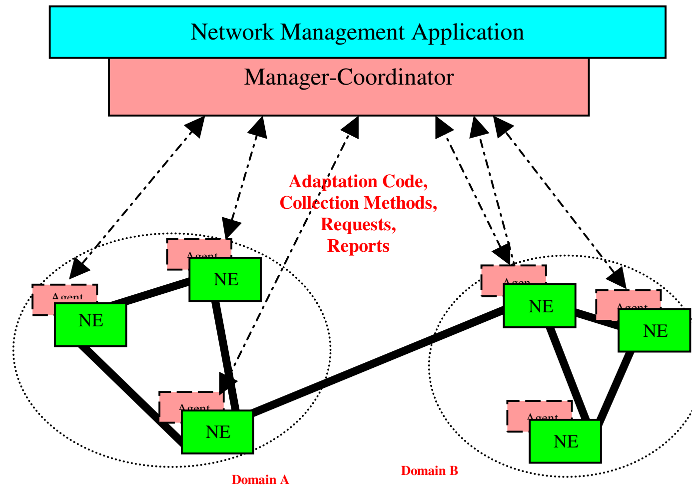

Figure 3-1: Components of Network Monitoring System

The dynamic control of the monitoring system is based on decision making and knowledge embedded in the network manager-coordinator. This enables the manager-coordinator to take decisions that re-direct the data collection, change the logic of the processing within the elements, and even direct the elements to execute tests. Similar level of intelligence is embedded in the fault and performance management applications [7,8].

4. System Design Issues

The design consists of facilities that allow a manager to collect data from the network element in such a way that changes to the collection method can be made at run-time. And, it allows run-time extension of the network element behavior through callbacks.

Agent Management

From the point of view of the manager, it is important to be able to manage the delegated agents. Management of the agents includes deploying and terminating them. It is also essential that the agents be able to send messages back to the manager. The manager should be able to send commands to the agents as well, perhaps to adjust some parameter of the agents. Figure 4-1 identifies the agent management functions.

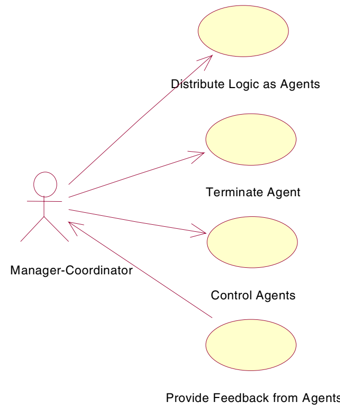

Figure 4-1: Agent Management

Agent Functions

From the point of view of the deployed agents, being able to read values and write values from and to the network element is critical. Also, it is necessary for the agents to be able to collect data across arbitrary data structures. Navigating these data types, such as queues and hashtables, could usually be performed through some native API associated with the abstract data type in the native processes. For this purpose, we shall include a facility to allow agents to call functions defined in the network element itself. Figure 4-2 identifies the main use cases that the agent performs.

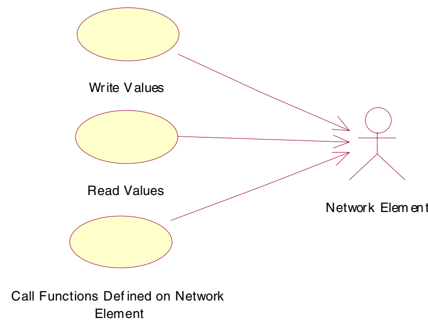

Figure 4-2: Agent Functions

Collection API

To enable the above functions, we need an interface between the network element native software and the delegated agent, with which the agent can define the specific set of resources that are considered useful to be monitored. The collection API must be able to support a range of data structures like queues and hash-tables.

If we know the address of any variable, we can read or assign its value. Of course, this implies that we know the type of variable we are dealing with, which probably requires the source code to be available. To access variables in such a way requires that we can find the addresses of the variables of interest. This is accomplished by examining the symbol table of the compiled code.

For the symbol table examination to work, we require that there is some way to extract addresses from the compiled code. For example, Solaris UNIX provides an 'nm' (*name mangle*) utility that allows the listing of symbols in an executable. We could use such utilities to create a directory of variables. A directory service will provide variable lookup by name; it also includes addresses of functions and function pointers.

On Extensions and Callbacks

As we claimed above, we could dynamically extend the functionality of the delegated agents or even the native software. The awareness of the need for extension is from human, not from the delegated agents or native processes themselves. It is the administrator again that determines the functionality of the added codes.

For the delegated agents, since the manager has full control of the agents' lifecycle, it is always a good option for the manager to create new agents with appropriately added functionality to replace the old ones by killing the old agents and deploying new agents. All the development is at the manager site and there is no need to recompile any code as long as the network elements support the same Virtual Machine as the management station does.

For native software, on the other hand, we don't have the luxury to put extra code off-line onto the original code at the manager site and deploy to network elements as a whole piece, without any recompilation. Here, the native code was already compiled and fixed; what we can do at the manager site is only to design and deploy the added code in its own fragment. We need a way to make sure that this added code could cooperate with the original code to have expected performance.

First, *callback hooks* are defined at certain places in the native code where the developer is reasonably suspicious that additional functionality may be needed later, but what he/she does not yet know at the time of software development. Defining a hook means putting an empty function at a certain place in the native code. Such hooks represent the possible places to add new functionality and they are the only locations where additional functionality could possibly be integrated.

If callbacks are determined as needed by administrator, the defined callbacks will be deployed and dynamically linked into the network element code by replacing the corresponding empty function at appropriate hook. Obviously, we need access to the function pointers in order to achieve the function replacement. Such function pointers could be obtained by the directory service as described above. Then, in the native software, this added code would be executed the next time this particular hook is encountered. Figure 4-3 illustrates these ideas.

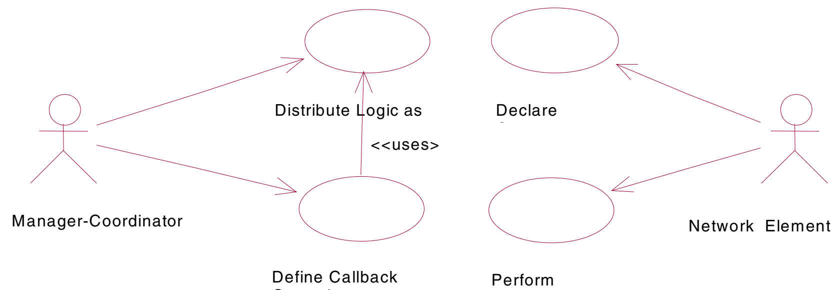

Figure 4-3: Callbacks

5. System Designs

5.1 System Designs based on Java Technology

Java Virtual Machine (JVM) provides the uniform infrastructure and native distributed programming API through Remote Method Invocation (RMI). In our Java based designs, agents are Java codes created at the Manager site and deployed to the network elements via RMI.

5.1.1 Java-based system design with MIB

In our first design here, we focus on legacy systems where SNMP is used. The SNMP agents collect raw data from network elements and store those data into Management Information Base (MIB). Our work here is to design a Java-based Extensible Management Server (JEMS) that runs as a server process at the network elements to host the delegated Java agents, which in turn, could access the MIBs and carry out their predetermined functionality. Figure 5-1 depicts the architecture of JEMS.

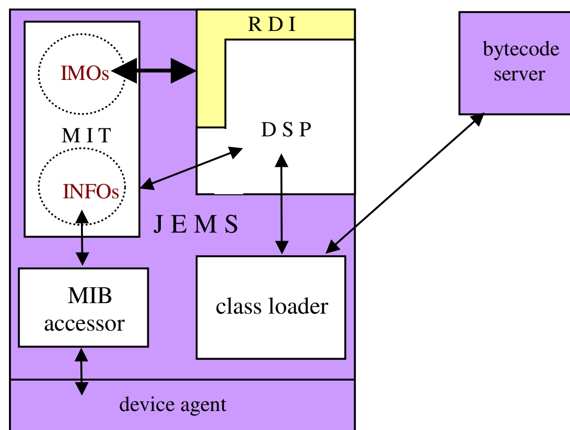

Figure 5-1: The JEMS Architecture

Remote Delegation Interface (RDI) is the interface through which management applications can remotely delegate Java objects, exchange information with these objects and control their execution. Management Information Tree (MIT) is a container that holds Java objects in a tree structure. Two kinds of objects are stored in the MIT. Intelligent Management Objects (IMO) are delegated by remote managers to the JEMS; they perform monitoring and control functions, and interact with remote managers via the RDI. INFormation Objects (INFOs) store management information in an object-oriented way; they are used by IMOs to implement management functions. Delegation service Provider (DSP) is an RMI server object that uses the MIT and the class loader to implement the RDI; it provides the delegation service needed by remote managers to delegate, control and communicate with IMOs. Class loader is an internal Java object used by the DSP to load, either locally or from some class server (a.k.a. bytecode server) on the network, those classes that are needed to instantiate corresponding delegated objects. MIB Accessor is another internal Java object used by INFOs to exchange low-level management information with the local MIBs.

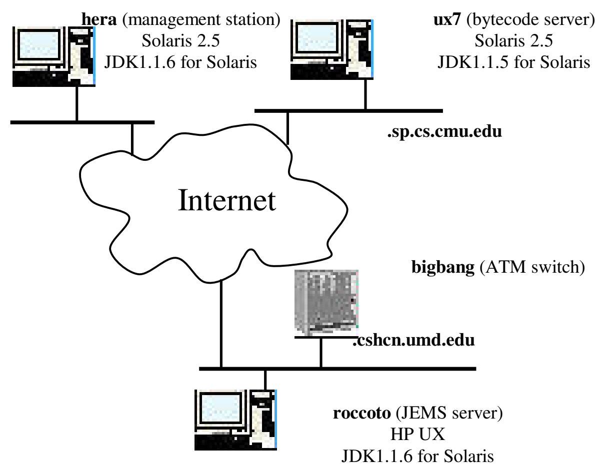

Figure 5-2: Prototype JEMS System

Based on the architecture described above, we built a proof-of-concept prototype system, as shown in figure 5-2. The managed device is a Fore ATM switch, which is connected to a LAN. It has a built-in SNMP agent that serves requests for variables defined in two MIB files: RFC1213 for IP management, and Fore-Switch-MIB for ATM-switch-specific management. According to the proposed architecture, a JEMS server should be running in the switch. However, since there is no JVM ported to the Fore ATM switch yet, we have to run a JEMS server in a workstation which is equipped with a JVM and acts as a *proxy* for the ATM switch. We have implemented some monitoring objects and conclude that JEMS provides a simple and flexible model to construct management systems, by allowing dynamic creation, manipulation and integration of delegated agents. Our system has better scalability, performance and

online extensibility than centralized polling systems. For more information about this system design, we refer to [17].

5.1.2 Java-based system design without MIB

The previous design is suitable for network elements equipped with MIBs, i.e. routers and switches. In many cases, however, network elements are not equipped with such MIBs. Even worse, these network elements may be equipped with only minimum amount of memory and computing resources, i.e., VSAT terminals for satellite communication networks. This Java-based design and the next native code based design are for such situations. One important issue of using Java in this case is that the network element side JVM has to be lightweight. We simply could not assume that we could ship the whole suite of the standard JVM down there. Specific versions of JVM are needed. For example, if the real-time operating system of the network elements were VxWorks, Personal Java suite was ported onto VxWorks. Another alternative is the so-called KVM [15], or Kilobyte Virtual Machine, that encapsulates only the core JVM and APIs. Such a KVM suite would typically require about 150KB memory, which is not stringent.

As to the collection API, Java Native Interface (JNI) could be utilized to help the communication between Java agents and the native C/C++ processes embedded in network elements. With addresses of functions and function pointers, we can provide callbacks and function replacement. For a function that is accessed through a function pointer, we can replace the function, by simply redirecting the pointer to our own replacement code. In the same way, we can provide a callback service. A function pointer can be declared in a native process, which calls this function pointer whenever it wants a callback. The function pointer in the beginning points to a null operation. Only when the Java code replaces the function will the callback be ready. Finally, we come full circle with the communication between Java and the target process by allowing a Java client to call functions from the target process. As mentioned previously, the directory service also has addresses of functions and the Java client can call these functions using JNI and the function addresses.

In summary, we use global variables as shared memory. So any variables of interest to the Java client should be global, and any functions to be replaced should be accessed through a global function pointer. Function pointers should be declared as a hook for us to create callbacks. Finally, we assume there is some way of extracting memory addresses from the symbol table similar to the name mangle utility in Unix. Figure 5-3 illustrates the logical view.

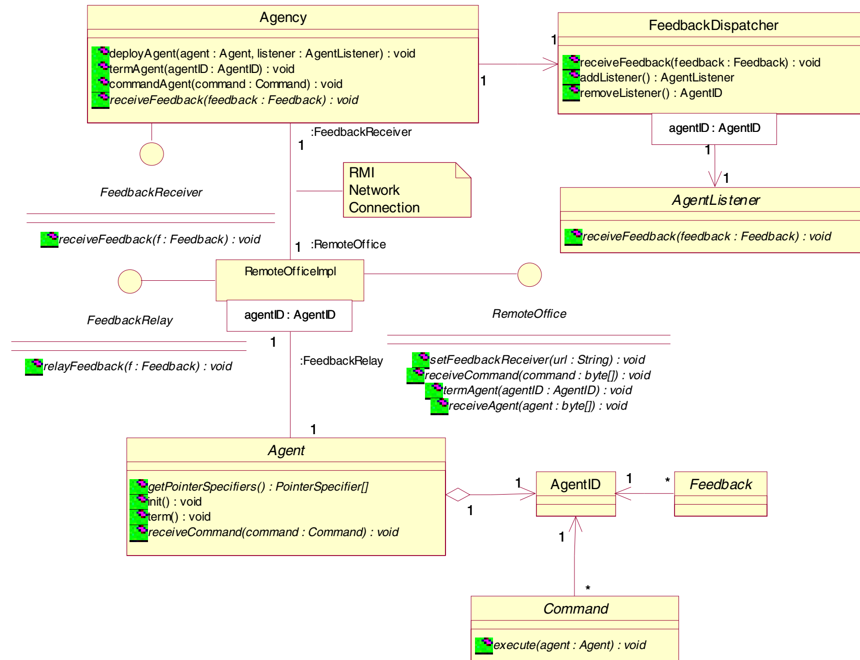

Figure 5-3: Java-based system design

5.2 System Design based on Native Code Technology

The last design based on Java technology assumes that the global variables and the processes are in a common address space. If, however, the network elements have multiple processors and separate address spaces, we have a different situation. Simply delegating Java agents to each address space would incur little extra work, but if we wish to change the native processes' logic and do the callbacks, things get worse. In this situation, the only way for a native C/C++ code to perform a callback onto Java is for a JVM to be running on each processor. This dose not seem practical.

In order for Java to interact with native C/C++ processes, it is necessary to use JNI, which incurs a great deal of overhead. This interface is quite limited and not particularly easy to use.

A native C/C++ implementation has neither the multiple JVM burden, nor the overhead and difficulty of a JNI implementation. And here we use native C/C++ codes in the system design. In order to read, write or call functions, the ordinary dereferencing of the pointers or function pointers will suffice, assuming that the Agent has the correct addresses in the memory of the network element.

This design requires Inter Process Communication (IPC) in two distinct places. One is between the network element and the management site, and the other is between the processes running on the network element in different address spaces. For the former,

it seems sensible to use Remote Procedure Call (RPC) or just an ordinary TCP/IP socket. The IPC between processes on the network element however, should not use sockets, which have much more overhead than required. We use shared memory based message queues with some semaphores to handle the IPC on the network element.

The network element code links the Agent code dynamically, so the network element resolves Agent symbols dynamically through ordinary dynamic linkage mechanisms. However, the Agent code does not have a dynamic mechanism to support the lookup of symbols in the target process. The solution to this problem is to look at the statically defined symbol table of the process residing in the executable code. As stated above, Solaris UNIX provides an 'nm' (*name mangle*) utility that allows the listing of symbols in an executable. Since the Agent code shall be defined in C/C++, we can provide a feature that takes the output of 'nm' and construct a directory based on it. Figure 5-6 illustrates the logical view of our native code design.

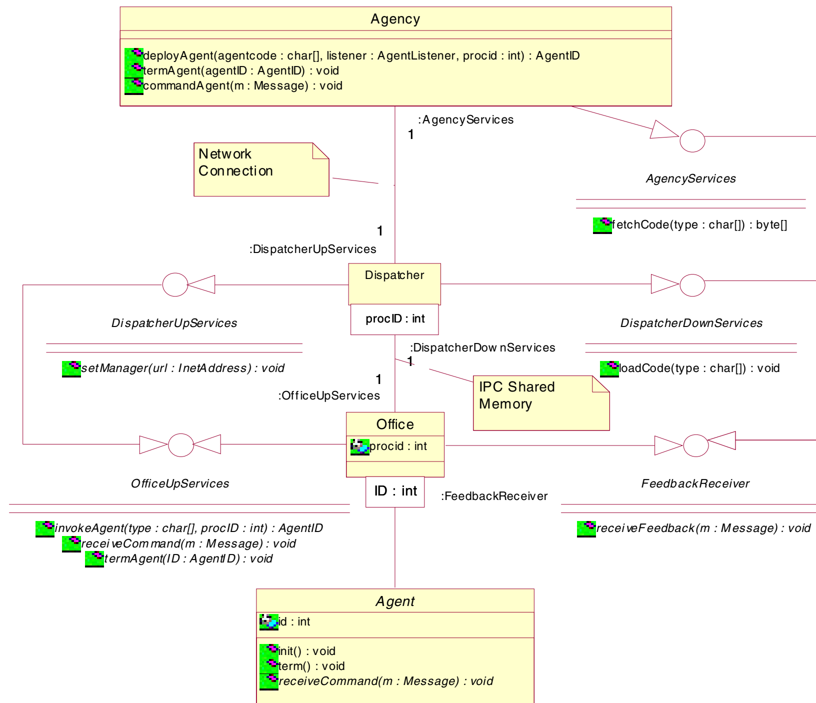

Figure 5-6: Native code based system design

5.3 Trade-offs between Java and Native code designs

During the processes of our systems designs, we encountered many issues that we need to balance between various design options. Choosing Java or native code technology is

one of the most important considerations. Here we list the advantages and disadvantages of using Java or native code for the system designs.

Table 1: Java System Design

Advantages	Disadvantages
▪ Distributed computing via RMI ▪ Code deployment and shipping via dynamic class loading ▪ JVM availability and portability ▪ Mobile computing support ▪ Exceptions handling	▪ One memory space assumption ▪ JVM memory footprint ▪ Callbacks are cumbersome ▪ Overhead of JNI ▪ Poor flexibility low-level synchronization primitives

Table 2: Native codes System Design

Advantages	Disadvantages
▪ Allows multiple processors ▪ Clear and neat design ▪ Lightweight callbacks via dynamic linkage ▪ No JNI overhead ▪ Flexible low-level synchronization primitives	▪ No RMI support; needs handle IPC explicitly ▪ No outsource dynamic class loading via HTTP ▪ No portability; it's rather a ad hoc design ▪ Explicit exceptions handling

6. Conclusions

In this paper, we have presented system designs for adaptive, distributed network monitoring and control. The focus of our work has been on three aspects. First, the design of the standard infrastructure, or Virtual Machine, based on which agents could be created, deployed, managed and initiated to run. Second, the collection API for our delegated agents to collect data from network elements. Third, the dynamic interface through which the functionality of the delegated agents or even the native software could be extended.

Our first design uses full-blown JVM in both manager and network element site and assumes the presence of MIBs. It is a proof-of-concept design and is suitable for network elements equipped with powerful computing and memory capabilities, i.e. routers and ATM switches. Here we use the off-the-shelf JVM and we do not need to access the network element native software directly; instead, we need only to access the MIBs that store the raw monitoring data. Our prototype system works well, which encouraged us to research further into the Virtual Machines and collection API issues.

In our second design, we consider the situations where there is no MIB embedded with network elements. We still use JVM but here our focus is on the network elements equipped with limited computing and memory capabilities. Specific versions of JVM are considered. For the delegated Java agents to access the native software, Java Native Interface (JNI) is exploited and a directory containing addresses of the native global variables and function pointers is set up. The processing logic of the delegated agents could be extended by creating new agents with the desirable functionality, followed by deploying them to the network elements to replace the old agents. To extend the native software functionality, we carry out function replacement by swapping the function pointers of the Java agents and the corresponding native code functions.

Further, in our third design, we remove the convenient JVM for those network elements equipped with multiple processors and address spaces. The focus here is to use dynamic linkage technology to emulate the Virtual Machine concept. The delegated agents are dynamically linked to the native code by the C/C++ run-time environment. The collection API in this case is very thin since all that is needed is to access the native code directly. The extension of functionality is similar as the second design, with the difference that we do not need JNI in this case. It is a neat design with respect to a pure C/C++ environment, but without JVM, it loses Java's portability. This design is suitable for those resource limited network elements that run over a real-time operating system and will not use Java as the native code development.

Now that we have the framework for adaptive, distributed network monitoring and control, our next step will be focused on the intelligence part of network management. In particular, we are interested in fault and performance management using such a framework, where the embedded intelligence would probably be domain knowledge, implementation of filters, execution of some tests, to name a few.

References

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

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

[3] A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, 24(5):342-361, 1998

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

[5] C. G. Harrison, D. M. Chess, and A. Kershenbaum, "Mobile Agents: Are they a Good Idea?", *Research Report*, IBM T. J. Watson Research Center, 1995

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

[7] H. Li, J. S. Baras and G. Mykoniatis, “An Automated, Distributed, Intelligent Fault Management System for Communication Networks”, *Proceeding of ATIRP'99*, 2-4 February, 1999
[8] H. Li and J. S. Baras, “Integrated, Distributed Fault Management for Communication Networks”, *Technical Report*, CSHCN TR 98-10, University of Maryland, 1998
[9] T. Magedanz, “On the impacts of intelligent agents concepts on future telecommunication environments”, in *Third International Conference on Intelligence in Broadband Services and Networks,* Crete, Greece, October 1995
[10] T. Magedanz, ``Intelligent Mobile Agents in Telecommunication Environments - Basics, Standards, Products, Applications'', *Tutorial for International Symposium on Integrated Network Management VI*, Boston, MA, May 1999
[11] J.P. Martin-Flatin. “Push vs. Pull in Web-Based Network Management”. In M. Sloman, S. Mazumdar, and E. Lupu (Eds.), *Proc. of the 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*, Boston, MA, USA, May 1999, pp. 3-18.
[12] G. Pavlou et al., “Distributed intelligent monitoring and reporting facilities,” *Distributed Systems Engineering*, vol.3, no.2, pp. 124-135.
[13] W. Stallings, SNMP, SNMPv2 and CMIP: the practical guide to network management standards, Addison-Wesley, Reading, Mass., 1993.
[14] W. Stallings, SNMP, SNMPv2 and RMON: practical network management, Addison-Wesley, Reading, Mass., 1996.
[15] Sun Microsystems, Inc, “Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices”, White Paper, Sun Microsystems, Inc, May 2000
[16] M. Wooldridge, N. R. Jennings, “Intelligent Agents: Theory and Practice”, *The Knowledge Engineering Review* 10(2) 115-152.
[17] H. Xi, Java-based Intelligent Network Monitoring, M.S. Thesis, CSHCN, 1999