On System Designs of Distributed, Extensible Framework for Network Monitoring and Control

by Hongjun Li, Shah-An Yang, and John S. Baras

CSHCN TR 2001-12
(ISR TR 2001-19)

On System Designs of Distributed, Extensible Framework for Network Monitoring and Control

Hongjun Li, Shah-An Yang, and John S. Baras

Center for Satellite and Hybrid Communication Networks

Department of Electrical and Computer Engineering

University of Maryland, College Park, MD 20742

{hjli, syang, baras} @glue.umd.edu

Abstract

In this paper, we present a distributed, extensible framework for supporting adaptive, dynamic network monitoring and control. We borrow the paradigm of management by delegation [8] and distribute some processing intelligence to network elements. The functionality of the delegated agents, and even that of the native software processes, could be extended dynamically without recompilation. Such procedure is called change of logic and we explain it in the framework of communicating finite state machines for extending native process functionality. We use Java technology and C/C++ dynamic linkage mechanism to achieve the standard hosting infrastructure for these agents and our system designs span a wide scope of applications.

Keywords

Network Management, Network Monitoring and Control, Mobile Code, Java, Extensible Agents, Management by Delegation

Submission area: Network Management Models and Architectures

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) [23,24]. Manager applications retrieve data from element agents by sending corresponding requests over the management protocol.

Such a system favors a centralized framework and works well for small networks. But as the networks become larger, more complex, and heterogeneous (e.g. multimedia networks), the centralized paradigm will incur vast amounts of communication between manager and agent and thus occupy too much bandwidth inefficiently. In this regard, we borrow the idea of Management by Delegation (MbD) [8] and distribute some of the processing logic and responsibilities by embedding code within the network elements. This embedded code within the network element is called a delegated agent.

In conventional network monitoring systems, the set of services offered by the element agents is fixed and is accessible through interfaces that are statically defined and implemented, for example Remote Monitoring (RMON) [27]. Statically pre-assigning functionality implies that the decision of what functionality to delegate needs to be taken at the agents' design phase. But, not all possible management tasks can be predefined this way. Further, requirements in the dynamic network environment change very often, which means that new type of functionality might be required now and then. 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, written in C/C++ for many cases, embody the processing logic viewed as necessary at the time of native software design and implementation. They may, however, lack consideration of 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, which usually incurs system down time, 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) [8] 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 [16,22]. The main advantage of this approach is that it is language independent. However, 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. Also, 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.

Flexible Agents and AgentX

In [21], Mountzia discussed temporal aspects of the delegation process and analyzed many issues concerning the application of the delegation concept in integrated network management. This framework is close to our system designs and it provided some helpful tips for our work. However, we also need to tackle the problem of extending the functionality of the native processes, which incurs many other issues, such as native collection API, callback mechanisms, etc.. In Internet community, RFC 2741 [5] defines a standardized framework for extensible SNMP agents. It defines processing entities called master agents and subagents, a protocol (AgentX) for the communication between them, and the elements of procedure by which the extensible agent processes SNMP protocol messages. RFC 2742 [11] defines the associated Management Information Base (MIB) that uses the AgentX protocol. In our work, however, we need to face such situations that there is no MIB embedded in network elements, i.e. small satellite terminals, and again, we tackle the problem of native process extension.

Mobile Agents

Another approach that enables dynamic downloading of functionality is provided by mobile agents [2,3,10,18,20]. Languages that are used to develop mobile agents include Java [9], Tcl/Tk [28], and Telescript [25], among others, and using mobile agents in decentralized and intelligent network management is a great leap from client-server based management pattern. Our system exploits the idea of code-on demand and focuses on mobile code [7] rather than mobile agents (in the sense of existence of an itinerary), still retains client-server architecture, and assumes a management server in each device concerned. Comparing with their mobile agent counterparts, the behaviors of our agents are much easier to understand and anticipate. Since our agents could also be implemented via native code, they are less straightforward, but more powerful.

Web-based Network Management

We are by no means the first people thinking of using Java technology in network management [12,19]. 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.

The rest of the paper is organized as follows. 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].

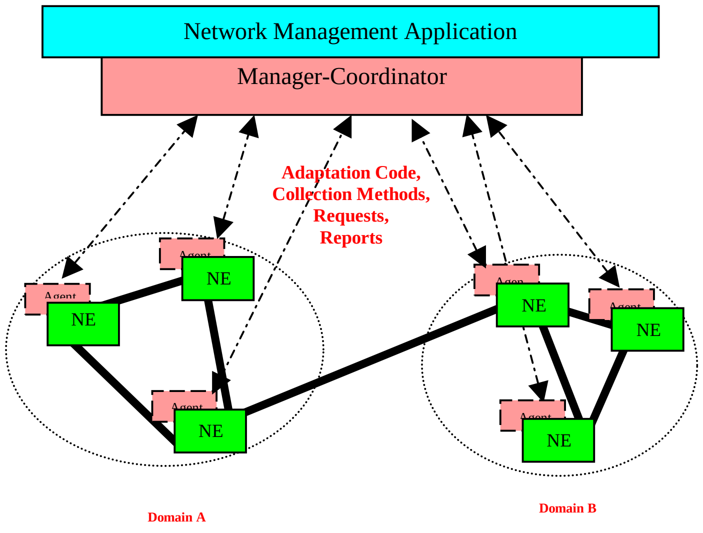


Figure 3-1: Components of Network Monitoring System

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 [13,14,15].

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.

- *Distribute Logic as Agents*: Sends agent collection logic across the network from the Manager-Coordinator to the network element. Must support dynamic linkage onto the network element.
- *Terminate Agent*: For an agent that has been deployed by invoking the *Distribute Logic as Agents* use case and still exists in the system, causes the Agent to be removed from the system.
- *Control Agents*: For agents that have been deployed with the *Distribute Logic as Agents* use case, send a command to the Agent instructing it to perform a generic action. The actual command to be performed will be determined by the specific command and Agent implementations.
- *Provide Feedback from Agents*: For an agent that has been deployed by invoking the *Distribute Logic as Agents* use case and still exists in the system, causes feedback to be sent from the agent back to the manager that created it.

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.

- *Read Values*: For an agent, which has been deployed using the *Distribute Logic as Agents* use case and has the address of what data it is looking to read, read values from the address space of the Network Element.
- *Write Values*: For an agent, which has been deployed using the *Distribute Logic as Agents* use case and has the address of what data it is looking to write, writes values into the memory space of the Network Element.

- *Call Functions Defined on the Network Element*: Suppose an agent that is going to do the actual function call has been deployed. Invoke a native C/C++ function defined in the Network Element.

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. This address extraction is also possible with standard dynamic linkage mechanisms as provided by VxWorks [29] and Solaris.

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 what functionality to add.

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.

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. Such hooks could be the places 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 could mean putting an empty function at a certain place in the native code. But theoretically, all functions in the native code could be thought of as suspects, even though we would usually not be so suspicious. 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 or dynamic linkage mechanisms as described above. Then, in the native software, this added code would be executed the next time this particular hook is encountered. Figure 4-1 illustrates these ideas.

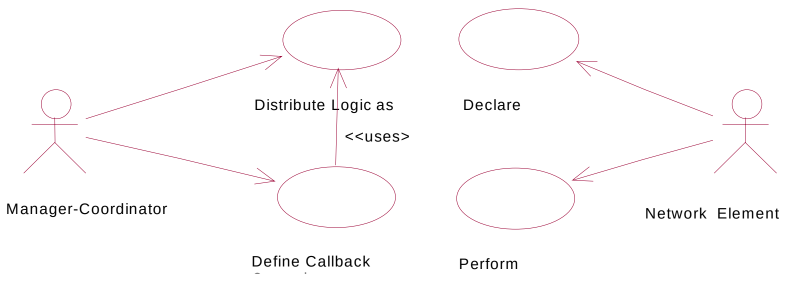


Figure 4-1: Callbacks

To better understand callback mechanisms, we model software processes using communicating finite state machines [4,17]. A finite state machine (FSM) consists of a finite set of states with one initial state, input set of the machine, and transition function, which is a partial function from the states and inputs to its states. An extended finite state machine (EFSM) introduces state variables in addition to the explicit states. These variables can take on a number of values themselves and they are treated as *implicit* states. The complete set of the states, or *global state*, of a process instance is now the union of the explicit and implicit states. When performing analysis, however, our focus is mainly on *system state*, which consists of explicit states, plus the status of enabled transitions from those states. The problem of state explosion is avoided as such.

Each native process is modeled as an EFSM. Each process instance has its own memory space that is under its own control. No other process instances are allowed to change the values of its variables. Such variables are called local variables. Local variables are further categorized as either state variables or temporary variables. State variables are those implicit states that represent the information accumulated and retained by a process. State variables are persistent, meaning that from the perspective of each process, they retain their value over time. One example is the variable that captures some certain statistics of interest, e.g. number of packets in the queue. Temporary variables are used to store information that does not require persistence. For example, an integer variable used as the index of a loop is typically treated as a temporary variable.

Apart from local variables, we also define some variables that are visible to all process models and they are called shared variables. Shared variables could be typically implemented via header files, in a C/C++ programming environment, and they are used as a communication mechanism among the process instances. Another way to model inter-process communication is message passing via input buffers. The management procedure of the input queue could also be modeled as an EFSM and to this end, we claim that our communications between EFSMs are all through shared variables.

There are no clear rules on the use of explicit states and state variables, as this is often a matter of design and depend very much on particular application. In our software process modeling, we use explicit states to represent the top-level modes or stages that a process can enter. Such a mode could be any waiting or inactive status, or it could be a decision place that leads to different situations. To facilitate callbacks, we also identify those places where some unanticipated situations might happen and where we might later put added functionality. We call these places callback hooks. Specifically, we associate each callback hook with two states: one is called pre-callback state and the other post-callback state. There is a transition from pre-callback state to post-callback state, with TRUE as the predicate and the callback function as the *action* associated with this transition. At the time of software process design, the callback functions could be just empty functions, and the pre-callback state and post-callback state look identical in the sense that all the accessible local/shared variables are the same. Or, as mentioned above, the callback function within the native code could be any function. In this case, the pre-callback state and post-callback state are not identical any more. Right after each post-callback state, we put a decision state to accommodate the possible multiple branches the process may lead to. The different branches defined over local/shared variables are mutual exclusive and exhaustive.

The parameters in the declaration of such empty functions are visible to an external entity. Symbol table examination via directory service or dynamic linkage mechanisms, as discussed above, provides a scheme to access the shared variables and the stack. The external callback function will use these local/shared variables to fulfill some added logic, and probably, some changes will be made on them since the function call is by reference. After this added external function is executed, the post-callback state will usually be different from that before this external callback function execution, in terms of the enabling branches from the decision state that follows. Such a mutual exclusive and exhaustive decision state ensures that there will always be a valid progress route for the process to move ahead, with or without addled logic.

Conceptually, our callback mechanism is similar to the idea of protocol boosters [6]. It is a supporting agent and by itself is not a process or protocol. Beyond protocol boosters, it handles some unanticipated situations. Many callback places in a process model may use the same function call, like DoCallbacks (CallbackID, shared_variables, state_variables, temporary_variables), and we assign for each callback hook a unique CallbackID for identification. To make sure that the augmented system work well, for both original and new conditions, we need to investigate some certain syntactic and semantic properties using system state analysis, based on the identification of states as discussed above. Such issues are beyond the scope of this paper.

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 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.

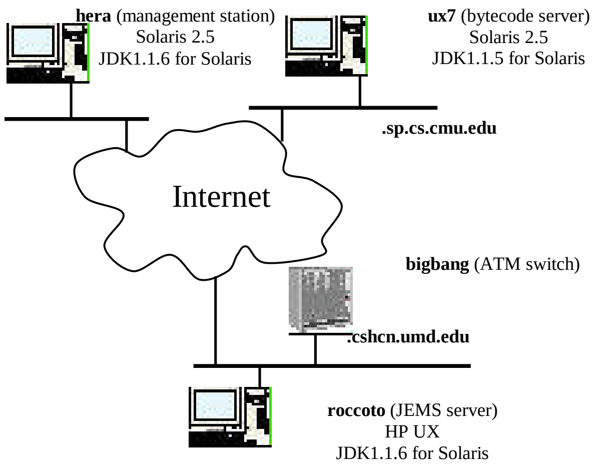


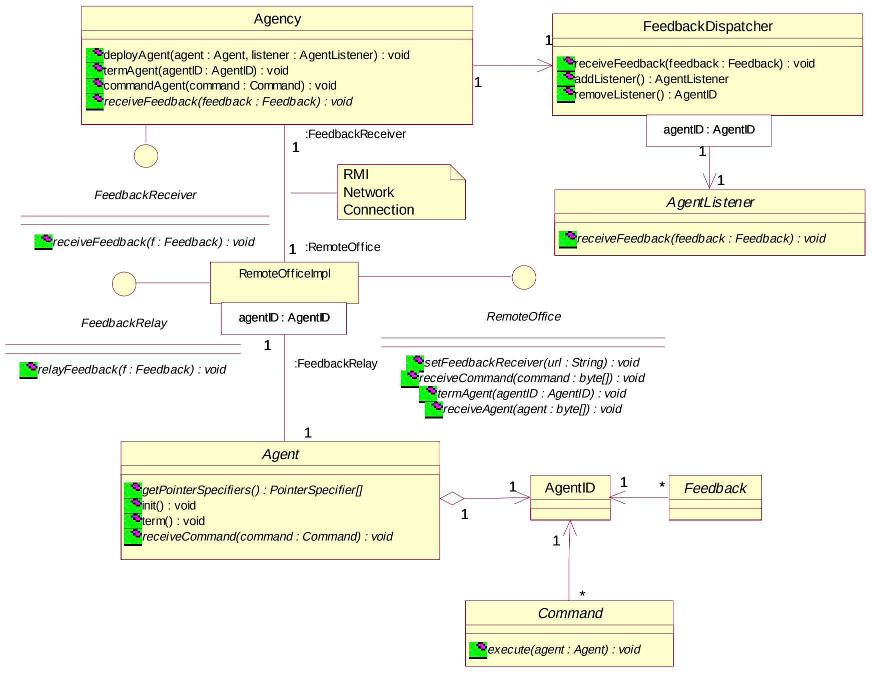
Figure 5-1: Prototype JEMS System

Figure 5-1 illustrates our proof-of-concept prototype system. 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. There is no JVM ported to the Fore ATM switch yet, and 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 [30].

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, e.g. 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 [29], Personal Java suite was ported onto VxWorks. Another alternative is the so-called KVM [26], 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. Figure 5-2 illustrates the logical view.



Agency
deployAgent(agent : Agent, listener : AgentListener) : void
termAgent(agentID : AgentID) : void
commandAgent(command : Command) : void
receiveFeedback(feedback : Feedback) : void
FeedbackDispatcher
receiveFeedback(feedback : Feedback) : void
addListener() : AgentListener
removeListener() : AgentID
agentID : AgentID
AgentListener
receiveFeedback(feedback : Feedback) : void
FeedbackReceiver
receiveFeedback(f : Feedback) : void
:FeedbackReceiver
RMI Network Connection
:RemoteOffice
RemoteOfficeImpl
agentID : AgentID
FeedbackRelay
relayFeedback(f : Feedback) : void
RemoteOffice
setFeedbackReceiver(url : String) : void
receiveCommand(command : byte[]) : void
termAgent(agentID : AgentID) : void
receiveAgent(agent : byte[]) : void
:FeedbackRelay
Agent
getPointerSpecifiers() : PointerSpecifier[]
init() : void
term() : void
receiveCommand(command : Command) : void
AgentID
Feedback
Command
execute(agent : Agent) : void
1
*

Figure 5-2: Java-based system design

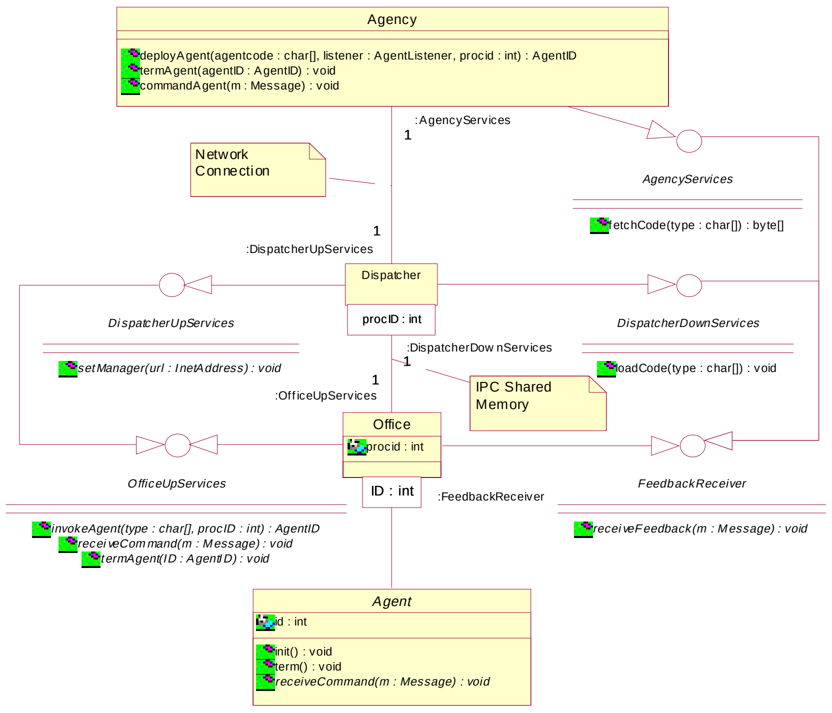
5.2 System Design based on Native Code Technology

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 does not seem practical. Also, 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. Once again, this could be done via dynamic linkage mechanisms. Figure 5-3 illustrates the logical view of our native code design.



Agency
deployAgent(agentcode : char[], listener : AgentListener, procid : int) : AgentID
termAgent(agentID : AgentID) : void
commandAgent(m : Message) : void
:AgencyServices
1
Network Connection
AgencyServices
fetchCode(type : char[]) : byte[]
1
:DispatcherUpServices
Dispatcher
procID : int
DispatcherUpServices
setManager(url : InetAddress) : void
DispatcherDownServices
loadCode(type : char[]) : void
:DispatcherDownServices
1
1
:OfficeUpServices
IPC Shared Memory
Office
procid : int
ID : int
:FeedbackReceiver
OfficeUpServices
invokeAgent(type : char[], procID : int) : AgentID
receiveCommand(m : Message) : void
termAgent(ID : AgentID) : void
FeedbackReceiver
receiveFeedback(m : Message) : void
Agent
id : int
init() : void
term() : void
receiveCommand(m : Message) : void

Figure 5-3: 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 ▪ Inflexible 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 a distributed, extensible framework for supporting adaptive, dynamic 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 communicating finite state machine based callback mechanism 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. An important advantage of this design is that large amount of data can be processed quickly via this native code callback mechanism, as compared with Java-based designs.

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. For more information, see our previous publications [13,14] and a sister paper submitted to this conference [15].

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. Bieszczad, B. Pagurek, T. White, “Mobile Agents for Network Management”, *IEEE Communication Surveys,* Vol. 1, No 1, pp. 2-9, September 1998.

[4] D. Brand and P. Zafiropulo, *“On communicating finite-state machines”*, Journal of the ACM, 2(5):323--342, 1983.

[5] M. Daniele, B. Wijnen, M. Ellison, Ed, D. Francisco, Ed., *Agent Extensibility (AgentX) Protocol*, RFC 2741, 2000.

[6] D.C. Feldmeier, A.J. McAuley, J.M. Smith, D.S. Bakin, W.S.Marcus, and T.M. Raleigh, “Protocol Boosters”, *IEEE Journal on Selected Areas in Communications,* Vol. 16, No. 3, April, 1998.

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

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

[9] J. Gosling and H. McCuilton, *The JAVA Language Environment (White Paper),* Sun Microsystems, Inc. 1995.

[10] 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.

[11] L. Heintz, S. Gudur, M. Ellison, Ed., *Definition of Managed Objects for Extensible SNMP Agents,* RFC 2742, 2000.

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

[13] H. Li and J. S. Baras, “Integrated, Distributed Fault Management for Communication Networks”, *Technical Report,* CSHCN TR 98-10, University of Maryland, 1998.

[14] 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.

[15] H. Li, S. Yang, and J. S. Baras, “A Framework for Supporting Intelligent Fault and Performance Management for Communication Networks”, submitted to *IFIP/IEEE International Conference on Management of Multimedia Networks and Services,* Chicago, October 2001.

[16] Luderer G., Ku H., Subbiah B., Narayanan A., "Network Management Agents supported by a Java environment", *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management (IM'97),* 1997.

[17] G.M. Lundy and R.E. Miller, “Specification and analysis of a data transfer protocol using systems of communicating machines”, *Distributed Computing,* vol. 5, pp. 145-157, 1991.

[18] 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

[19] 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.

[20] Mobile agent technology, General Magic, Inc., http://www.genmagic.com/agents

[21] M. Mountzia, “A distributed management approach based on flexible agents”, *Interoperable Communication Networks*, 1 (1998) 99-120

[22] G. Pavlou et al., “Distributed intelligent monitoring and reporting facilities,” *Distributed Systems Engineering*, vol.3, no.2, pp. 124-135, 1996

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

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

[25] Telescript Technology: the foundation for the electronic market-place, General Magic Inc., http://www.genmagic.com/agents/Whitepaper/whitepaper.html, 1996

[26] Sun Microsystems, Inc, "Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices", White Paper, Sun Microsystems, Inc, May 2000

[27] S. Waldbusser, *Remote Network Momitoring Management Information Base,* RFC 1757, 1995

[28] B. Welch, *Practical Programming in Tcl and Tk*, Prentice-Hall, 1995

[29] VxWorks, http://www.windriver.com

[30] H. Xi, Java-based Intelligent Network Monitoring, M.S. Thesis, CSHCN, 1999