CERN-IT-2000-005
19 May 2000

The Java Management Extensions (JMX): Is Your Cluster Ready for Evolution?

Javier Jaen-Martinez
Physics Data Processing Group
Information Technology Division
European Laboratory for Particle Physics (CERN)
CH 1211 Geneve 23 Switzerland

E-mail: Javier.Jaen-Martinez@cern.ch

Abstract

The arrival of commodity hardware configurations with performance rivaling that offered by RISC workstations is resulting in important advances in the state of the art of building and running very large scalable clusters at "mass market" pricing levels. However, cluster middleware layers are still considered as static infrastructures which are not ready for evolution.

In this paper, we claim that middleware layers based on both agent and Java technologies offer new opportunities to support clusters where services can be dynamically added, removed and reconfigured. To support this claim, we present the Java Management Extensions (JMX), a new Java agent based technology, and its application to implement two disjoint cluster management middleware services (a remote reboot service and a distributed infrastructure for collecting Log events) which share a unique agent-based infrastructure.

KEYWORDS: Cluster Management, Software Agents, JDMK, JMX

1.1. Getting Help

According to Pfister [1] there are three ways to improve performance: work harder, work smarter, and get help. The analogy in terms of computing technologies [2] is that working harder is like using faster hardware, working smarter concerns using more efficient algorithms and techniques, and finally getting help refers to using multiple processing entities to solve a particular task.

There are several architectures that can accommodate the “getting help” approach depending on how processors, memory and interconnect are laid out. The most relevant examples over the past two decades are: Massively Parallel Processors (MPP), Symmetric MultiProcessors (SMP), Distributed Systems, and Clusters (see [3] for a detailed comparison of the architectural and functional features of these systems).

Since the early 90s, the rapid improvement in the availability of commodity high performance components for computers and networks are making networks of computers an appealing vehicle to move away from expensive and specialized proprietary supercomputers. Moreover, clusters are easy to integrate into existing networks and nodes can be easily grown adding memory or additional processors.

Clusters are designed to appear as unified resources, in other words, they must have a Single System Image (SSI). A Middleware layer located between the Operating System and user-level environment supports the SSI. This layer consists of two components of software infrastructure: the system single image infrastructure (SSII) and the System Availability Infrastructure (SAI). The SSII glues together operating systems on all nodes to offer a unified access to system resources, whereas the SAI enables the cluster services of checkpointing, automatic failover, recovery from failure and fault-tolerant support among all nodes of the cluster [2].

Traditionally, *cluster management software* (CMS) involved any software designed to administer and manage application jobs submitted to clusters. This approach encompassed the traditional batch and queuing systems [4, 5]. More recently, CMS has been generalized to cope with additional middleware functions but clusters are still considered as *static* infrastructures whose middleware layers are not ready for evolution. Unfortunately, this assumption is not anymore valid in environments where the distributed architecture of today might have to evolve to cope with the requirements of tomorrow. In this context, rebuilding from scratch our middleware layers is not an effective way to deal with evolution. Instead, we should start to establish the requirements for a foundation layer that is well designed for this purpose.

In this paper, we claim that middleware layers based on both agent and Java technologies are ready for evolution and can support clusters where services can be dynamically added, removed and reconfigured. To support this claim, we present a new Java agent based technology and its application to implement two cluster management middleware services: a remote reboot service and a distributed infrastructure for collecting Log events. These two services will use the same underlying agent infrastructure proving that this technology, once it is deployed, can accommodate dynamically different types of middleware services.

2. JMX: JAVA AGENTS FOR DYNAMIC MANAGEMENT

2.1. Why Agents?

Software agents have been traditionally related to research on computational models for distributed artificial intelligence (AI) [6, 7, 8, 9]. Under this point of view, agents are seen as entities that accept a user's set of goals and carry out in an effective way the tasks for these goals to be achieved. Agents seen as intelligent entities have been clearly oversold beyond present technology (only very simple examples and limited intelligence has being proven). However there are still some non-intelligent features of software agents that make them a very interesting approach to solve two practical problems: the inherent complexity of managing resources in highly distributed environments, and the limitations of user interface approaches based on direct manipulation [10].

First, to understand how agents contribute to simplify management of distributed resources it is a very good exercise to have a look at how distributed computing has evolved in the last decade. As pointed out by [10] a few years ago, most software configurations exhibited only basic mechanisms of cooperation (e.g. file transfer, print servers, and database queries). Later on, a few important standards like TCP/IP brought connectivity to applications but still adhoc/proprietary application level protocols had to be in place to achieve intelligent cooperation among different systems. With the growing success of object oriented technologies this problem was addressed by more encapsulated communication mechanisms like the ones provided by different technologies: CORBA, COM, and RMI. These technologies certainly allow a collection of cooperating distributed processes to interact in the same way as if they were running in the same machine. However, applications for distributed resource management built on top of them are "static" in the sense that they can only work with resources (applications, devices, services, etc) that are known in advance by all the elements of the management system. There are solutions for resource management built on top of COM and CORBA that provide more dynamic applications but we are still far away in these technologies from having an standard mechanism for COM components or CORBA servers to add and publish dynamically new services. For instance COM forces a new component to be created and registered within the registry when new interfaces must be supported which is a less flexible mechanism than that provided by technologies like JMX as we will present later. Therefore, a distributed system where services are created, removed, reinstalled, reconfigured, and activated depending on rapidly changing demands cannot be managed by a static management system simply because resources are updated faster than the system's implementation can be updated. Agents, on the contrary, are dynamic in nature. In the AI domain, agents' capacity to integrate evolving and changing behaviors was a mandatory requirement since agents were conceived as task oriented entities to implement task specific solutions (as opposed to more traditional AI approaches where systems were modeled as general and domain-independent as possible) [9]. This important requirement is not only useful for intelligent domains but also has been successfully applied to simplify management of distributed resources. This feature allows building architectures in which each individual dynamic agent within a network implements a specific set of management functions with powerful evolution mechanisms to allow these functions or responsibilities to change over time.

Second, we will discuss how agents can overcome some limitations of interfaces based on Direct Manipulation (DM), a style of Human Machine Interaction design which features a natural representation of task objects and actions manipulated by people (directly) not through an intermediary. This will be also a point to take into consideration in favor of agent based technology to perform effective cluster management tasks. Using DM techniques, as opposed to command-line interfaces, results in interactive systems whose operation is easy to learn and use and difficult to forget (a psychological aspect of using visual representations) [11, 12]. However many of these advantages start to be questioned when the number or complexity of tasks in hand grows. Under such conditions, it is evident that a sequence of tasks can be better performed automatically than directly by tedious user manipulation of visual objects.

2.2. Why Java?

There are numerous advantages offered by software systems based on Java technology: platform independence, multithreading, re-use opportunities through a component based model (JavaBeans), and support for distributed objects (among others). However, many critical opinions have also traditionally arisen pointing out one of Java's main drawbacks: lack of performance when compared to languages like C/C++. Some empirical benchmarking studies [13, 14] indicate that C++ is on average seven to eight times faster than Java (with specific results being even more disappointing on the Java side). These results should be carefully taken into consideration by analyzing the circumstances and conditions under which experiments were performed. Usually, not the best JVM implementations were considered and no optimization techniques were put in place in order to obtain more objective conclusions. In fact, as shown in [13, 15] the choice of virtual machine has a great impact on performance and the use of techniques like Just in Time compilers makes Java a lot more competitive for a lot more applications. Therefore, it is a fact that acceptable levels of performance can be obtained with a sensible choice of the underlying virtual machine and a good exploitation of adequate optimization techniques. JIT compilers, multithreading, reduced use of synchronized methods, understanding the cost of some operations on strings, function inlining, object reuse, among others, have to be properly used to obtain the best performance in Java based applications.

After all, nobody can deny that Java adds another level of overhead and abstraction in the same way that other layers (fortunately for those who had to develop applications in assembly code) have added overhead during the history of computing. The question is whether this overhead pays for itself with the additional functionality that is provided. We expect to show in the following sections and examples that Java together with Agent technology pays for its overhead by providing flexibility, extensibility, scalability, and superior design when designing management tools for scalable clusters.

2.3. The Java Management Extensions

2.3.1. Overview

JMX is a specification recently released by Sun Microsystems[16] that takes advantage of both Java's unique features and agent-based architectures to obtain management systems that have the following features:

- **Extensibility**: JMX agents can dynamically incorporate new functionality (Java Beans) from remote class servers or by taking advantage of existing Web servers. This way, plug & play management environments can be designed where both resources and management applications are dynamically updated according to changing requirements.

- **Interoperability**: JMX agents can be integrated in TMN, SNMP, CIM/WBEM, and CORBA management systems.

- **Reusability:** JMX's management model is component-based. In other words, new management functionality (developed externally or available in the "software market" in the form of JAVA Beans) can easily be integrated in any existing JMX based architecture. Thus, JMX based architectures can benefit from the best available management solutions available for each existing resource in our system.

- **Platform independence:** JMX agents can be deployed in non-homogeneous environments where different platforms coexist.

Architecture

The JMX model is based on a three layered architecture (see Figure 1). This results in additional flexibility allowing subsets of the model to be used for specific purposes.

- **Instrumentation level**: This level provides a specification for implementing JMX manageable resources including applications, devices, users, services, etc.

- **Agent Level**: JMX agents are the containers where Mbeans are loaded, and handled. Each agent integrates at least one communication adaptor or connector that allows management applications to communicate with Mbeans using a particular protocol RMI, HTTP, IIOP, etc and a set of services to handle the beans within its framework. Thus, agents are implemented in a protocol independent way enhancing flexibility and reuse under different management contexts. Additionally JMX agents could be used as a very efficient mechanism to avoid duplicate virtual machines running on a single system by hosting any new set of beans to be executed on demand.

- **Manager Level**: this level provides a specification for implementing JMX managers that provide an interface for management applications to interact with agents through its connectors.

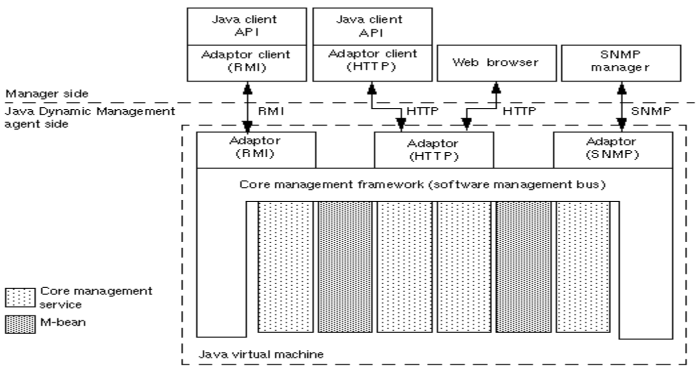

Figure 1. JMX three-level architecture

3. IMPLEMENTING CLUSTER MANAGEMENT FUNCTIONS WITH JMX AGENTS

JMX is not only a specification or a set of theoretical ideas on paper but also has been implemented as a set of Java packages integrated in a kit called the Java Dynamic Management Kit (JDMK) [17]. The first public release of the JMX specification was on November 1999 and the latest version of the JDMK (4.0) was released last December. Despite the recentness of this technology, we have had the chance to use JDMK 3.2 which integrates most of the features described in the JMX specification to implement several prototypes that prove this technology to be a serious candidate to be considered when implementing management solutions for clustered architectures. The following examples will illustrate how some of the features described above can be used to implement scalable management solutions.

3.1. Implementing a Remote Rebooting Service

3.1.1. Description of the problem

Let us consider a cluster consisting of a large number of machines (around 500) dedicated to process batch jobs. The cluster must provide services under both LINUX and Windows NT as a result of having 3 populations of users (two populations exclusively using either LINUX or NT based applications, and a third class of users with varying needs of these two types of OS). In such a context, the number of nodes running LINUX versus NT has to evolve dynamically according to the current workload of the system and user demand. Thus, an infrastructure that allows controlled remote rebooting of selected nodes of the cluster is required to avoid system administrators to be physically present to reboot and then type on a local keyboard the kernel image that has to be loaded after rebooting. This software infrastructure clearly increases the degree of flexibility and manageability of the cluster because the rebooting process of any number of nodes can be done remotely from a single console without human presence on the physical location where the cluster is installed.

3.1.2. JMX Infrastructure

3.1.2.1. Instrumentation Level

Our infrastructure requires the Linux Loader (LILO) [18] to be installed as a boot loader in the Master Boot Record (MBR) of all the nodes of our cluster. This simplifies very much the LINUX side of the problem, i.e. an Mbean in a JMX Agent that is running on a LINUX machine can easily make use of the LILO services to select the kernel image that will be loaded in the next reboot. On the contrary, implementing a bean that can, in the same way, select the next loaded image when the JMX Agent is running under NT is a great technical challenge. A complete and detailed description of how this can be achieved is out of the scope of this paper. To simplify, a bean running under NT first reads the MBR to obtain the address on disk of the physical sector where LILO stores both the name of the next image to be loaded and its initialization parameters. Second, transforms this encoded physical address into a standard logical block address (LBA), and third, writes on this block the adequate information for LILO (magic numbers, name of kernel image, and initialization parameters). For the scope of our discussion, let us assume that we are able to implement two Mbeans (LINUX and NT versions) that are able to: select the kernel image to be loaded, accept the boot command-line parameters, and trigger the reboot process within a specified time delay.

3.1.2.2. Agent Level

In our architecture each node hosts a JMX agent and each agent hosts a bean that provides the functionality described above. Additionally, there is a master agent that keeps track of the state of the nodes in the cluster. This way, the management application, when first started, does not make use of broadcast messages to detect which nodes are "alive" but instead contacts the master to obtain information about registered nodes in the cluster and their current status[1]: running NT, running a specific LINUX image, rebooting, or down. The subagents, consequently, contact the master when going UP (registration process) and DOWN (release process). Additionally, the master polls periodically[2] all registered subagents in order to detect unexpected node crashes or failures.

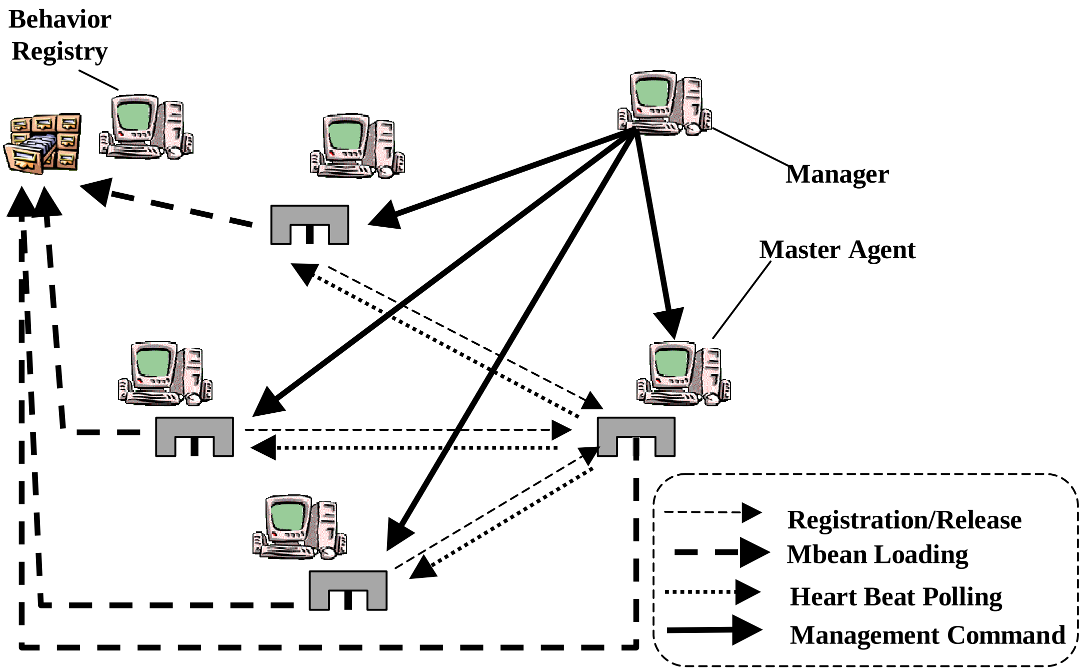

Figure 2. Agent Level: Communication schema

Once the general activity of the system has been described there are still several points to be discussed: How do JMX agents obtain the correct bean version (Linux or NT) that provides the rebooting functionality? What if the Master Agent crashes? How can Beans with enhanced functionality be deployed once the architecture is in place and running? How are beans remotely manipulated by a management application?

Mbean Loading

All agents (including the master agent) make use of a remote loading service (a JMX service) to obtain the Mbeans that will define their functionality from a remote server. An agent does not know in advance what beans have to be loaded. Instead, a set of bean configuration files is downloaded from a

[1] This could be avoided if the discovery service provided by JMX had been used. Unfortunately, our test network does not allow multicasting groups to be formed.
[2] The polling period can be set up by the management application

remote class server. The files to be downloaded (called MLET files in JMX terminology) describe with a tagged style similar to HTML the remote location, the name, and initialization values (among other information) of all beans that have to be created. The remote class server can be seen as a centralized configuration repository or as a "***behavior registry***" (Figure 2). In our current configuration, the *behavior registry* contains a generic MLET file for every node to be downloaded and specific ones so that customized configurations can be done on a "per agent" basis. For instance, node "*A*" running Windows NT downloads the generic MLET file located at *http://www.cern.ch/IT/JMX/**generic/init.txt,*** but also *http://www.cern.ch/IT/JMX/**A/init.txt,*** and *http://www.cern.ch/IT/JMX/**A/NT/init.txt***[3] defining its personalized configuration. This way, by downloading ***…/NT/init.txt*** or ***…/LINUX/init.txt*** respectively, every node obtains in a very straightforward and dynamic way the adequate version (NT or LINUX) of the rebooting Mbean.

Master Agent Robustness

This flexible mechanism can be also used to define dynamically which node will host a master agent. If node *A* has to be a master agent then the information to locate the bean that defines the "master behavior" will be added to ***…/A/init.txt*** and automatically[4] *A* will become a master agent. All subagents also know who is the master agent since this information is stored in ***…/generic/init.txt*** which is loaded at every startup, and later polled periodically. So, if the current master crashes (the point of failure of this architecture) the recovering mechanism is quite straightforward. Using the management application, a new Master agent *B* can be designated. The management application then updates ***…/generic/init.txt*** and ***…/B/init.txt*** and all subagents are automatically updated when consulting the *behavior* registry, initiating a new registration process with the new master. Note that the manual designation of a new master agent after a crash occurs can be automated with a simple protocol. The active master agent can refresh periodically (setting the current system time) a token located in ***…/generic/init.txt.*** If the master Agent crashes, this token is not refreshed. Each subagent downloads the token when contacting the behavior registry. If a subagent "*C*" detects an outdated token, then it locks the behavior registry (no other subagents will consult the registry meanwhile), names himself as the new master agent and updates conveniently ***…/generic/init.txt*** to let other agents know that a new master agent has been created[5].

Extensibility

The behavior registry (based on MLET files) is a very powerful versioning mechanism because just modifying this centralized software repository can reconfigure the functionality of the entire cluster. Agents, being "active entities", access periodically the repository looking for new versions or new beans to be incorporated into their frameworks which frees system managers from the burden of reconfiguring manually "passive daemons" as it is traditionally done in systems that do not use agent technology.

Protocol Independence

---

[3] As described earlier this is NOT the only mechanism available in JMX to obtain remote classes.
[4] Every agent consults its repository periodically to obtain any new behavior (mbean).
[5] This protocol requires the clocks of all nodes in the cluster to be synchronized and other subtle aspects should be taken into consideration for this mechanism to work properly.

Our current prototype was implemented with agents that are accessed remotely via RMI. So, our management application (see next section) makes use of this mechanism to get/set properties or invoke methods on Mbeans. However, being JMX protocol independent, it would possible to implement either a management applet running on a web browser that would manipulate our distributed architecture of agents via HTTP or even a CORBA client with IIOP to achieve the same goal. Of course, our architecture and agent implementations remain the same. The adequate communication adaptors (see Figure 1) will transform external HTTP/IIOP requests into internal method invocations.

3.1.2.3. Manager Level

The manager level (following the architecture defined by the JMX specification) is implemented as a Java application based on Swing components. The interface in this case consists of a simple tree and a context sensitive menu that is activated when a node on the tree is clicked. Of course our application is platform independent and has been tested in both Windows NT and LINUX environments.

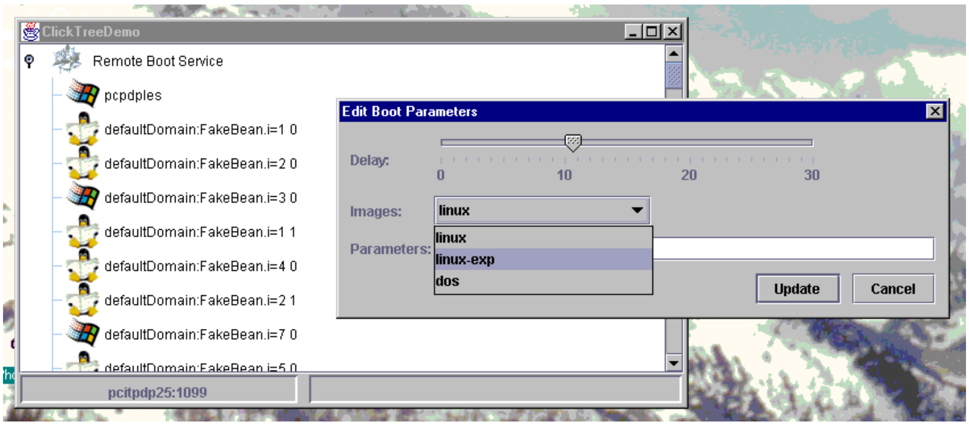

Figure 3. Management Application

In the example shown in Figure 3, only one node in the tree reflects a real machine *pcpdples* with three kernel images: *linux*, *linux-exp*, and *dos*. The remaining visible nodes were simulated during scalability tests. These tests were performed varying the number of subagents (*n*) between 0 and 300. The actual cluster contained only 2 nodes: one node hosts a master agent and a second node hosts a subagent with *n* Mbeans (simulating *n* real nodes). In these tests, the master agent was contacted simultaneously by both, the *n* Mbeans every 4 seconds simulating *n* registration/release operations, and the management application every 3 seconds to obtain the status information of the *n* nodes. The tests were successfully performed with the master agent running on a Pentium II (400Mhz) with Windows NT 4.0. The CPU utilization of the Master agent with and without the management application is shown in Figure 4. The obtained data shows that, in the range of nodes tested, the master agent scales linearly. It can also be observed that the existence of a management application polling periodically the master agent adds a constant overhead. Further experiments increasing the number of subagents will be performed and the influence of the OS in the scalability of the Master agent will be investigated. No data could be collected above 300 nodes because an RMI limitation was observed in the subagent that hosted the *n* simulation beans.

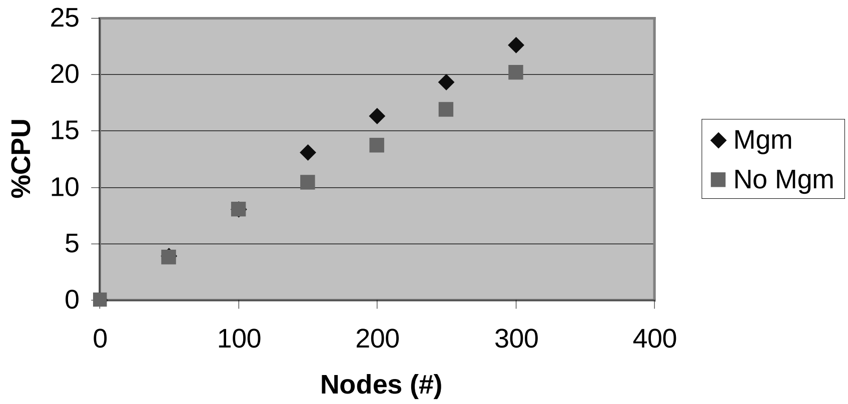

Figure 4. Master Agent CPU utilization vs number of nodes in the cluster

3.2. Implementing an Infrastructure for Collecting Distributed Log Events

3.2.1. Description of the problem

Let us consider a second management requirement that illustrates how existing JMX infrastructures can be dynamically extended with new services. In this case, our goal is to define a mechanism that allows instrumented processes to generate log messages that will be collected by a centralized "*collector*". A collection of instrumented processes together with a *collector* form a "collecting session". There can be multiple simultaneous *collecting sessions* and both the collectors and the instrumented processes may be located in any arbitrary node of our cluster. A collector must support sorting (events generated in multiple locations must be sorted according to their respective timestamps), filtering (recording only those events of interest within a session), as well as event notification (notifying a set of listeners upon arrival of certain events of interest). Additionally, the management application must have control over the running processes on a collecting session so that instrumented processes can be stopped, restarted, and observed (its standard output can be consulted from a single console). This new service will be used to eventually perform cluster tuning, alarm generation, distributed application debugging and performance evaluation.

3.2.2. JMX Infrastructure

3.2.2.1. Instrumentation Level

In this context we have two instrumentation levels: the instrumentation of processes to generate log events, and the definition of Mbeans that can behave either as collectors or as process controllers.

In order to instrument processes that can generate log events we will make use of a powerful package known as *NetLogger* developed at Lawrence Berkeley National Laboratory [19] by Brian Tierney and his team. *NetLogger* is a Toolkit through which C/C++, Java, Perl and Python processes can be instrumented to generate log messages which are in Universal Logger Message (ULM) format.

For the second instrumentation level we defined an Mbean known as *Remote Netlogger Bean* (RNB) that includes functionality to:

- collect, sort, and filter ULMs,
- generate alarms, and
- start, stop, restart and get the standard output of instrumented processes.

Thus, any JMX agent that integrates a RNB becomes an active entity performing all the previously described activities.

3.2.2.2. Agent Level

Our agent infrastructure is very similar to that described in the previous example. We use the same agents deployed there and the same behavior registry. Again, a master agent is needed as a repository for the collecting sessions that have been defined. To allow every agent in the cluster to get a copy of our RNB only the ***…/generic/init.txt*** file of the registry need be modified to include the description of the new RNB and again the same strategy is used to designate a master agent. Every agent periodically checks the registry in search of new functionality and the update process is achieved automatically without user intervention or manipulation of each agent on the cluster.

In this example, the crash of a master agent is a more critical factor because the master agent contains vital information that has to be recovered when a new master agent is designated. To solve this problem, the master agent makes use of a persistent repository service (a JMX service) which stores data in a central data repository. The location of this data repository is defined as an initialization parameter in the MLET file of the JMX agent that plays the role of master agent and the repository is deserialized any time a new master agent is created.

In the current version of our architecture, in case of a crash of an agent hosting a collector or a collection of instrumented processes, no recovering mechanisms have been defined. If the failing agent performed collecting operations, the affected sessions and instrumented processes are stopped and released in the master agent. If no collector is affected by the crash only the instrumented processes in the failing host are affected by the failure. Adding local persistent repository services to every agent in order to store locally on disk the data structures associated to a RNB will solve this weakness of the current design.

3.2.2.3. Manager Level

Our manager application contacts the master agent to get a list of the currently defined collecting sessions. This list includes information about the collector host and the location of the instrumented processes. Then, for each session, the manager contacts the RNB of the collector and the RNBs where instrumented processes exist for the session. This information is displayed as a tree where all different collectors are shown together with the instrumented processes. Once a session is started, all its instrumented processes are launched and the filtering and signaling mechanisms activated. To achieve event signaling we make use of event listeners as provided by the Java environment. Again context sensitive menus are used to manipulate both collectors and instrumented processes.

4. CONCLUSIONS AND FUTURE WORK

We have presented a novel Java technology (JMX) integrating Agent features. This technology is a very interesting approach to solve the problem of designing flexible middleware layers for clusters with evolving computing requirements.

Developing agent-based software architectures under JMX results in frameworks that are extensible, protocol-platform independent, reusable and interoperable with existing middleware services based on TMN, SNMP, CIM/WBEM, and CORBA. Services like persistent repositories, metadata, dynamic behavior loading, monitoring, alarm generation, and agent cascading are already integrated in this technology and ready to be used.

We have successfully used this technology to implement two prototypes of basic middleware services: the management of remote reboot operations on heterogeneous multi-platform clusters and the collection of log messages in a distributed set of instrumented processes using the same underlying agent-based infrastructure. This proves that dynamic management can be achieved in this approach by just configuring the functionality (java beans) that each agent must host. Additionally JDMK agents have proven its capability to host in a single java virtual machine a collection of completely disjoint services which can be accessed through a variety of protocols avoiding this way the waste of memory that results from having separate virtual machines running on a single host.

Future work will include developing a generic infrastructure for deployment of extensible middleware services based on JMX, and extending JMX functionality by implementing a Workflow service. This service will be the foundation for allowing complex user tasks to be defined and performed by agents in a distributed environment.

5. REFERENCES

[1] G. Pfister. *In search of clusters*. Prentice Hall PTR, NJ, 2nd edition, NJ, 1998

[2] R. Buyya. *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall, Melbourne, 1999

[3] K. Hwang, Z. Xu. *Scalable Parallel Computing: technology, Architecture, Programming*. WCB/McGraw-Hill, NY, 1998

[4] Jaen-Martinez , J. Farm Computing Issues and Examples. 3rd LHC Computing Workshop Marseille, France (1999)

[5] Jaen-Martinez J. PC Cluster Technology: State of the art and Future Challenges. Technical Report. Physics Data Processing Group. IT-CERN 1999

[6] Brooks R.A., Intelligence without Reason, Computers and Thought lecture, Proceedings of IJCAI-91, Sidney, Australia, 1991.

[7] Wilson S.W., The Animat Path to AI, In: From Animals to Animats, Pro- ceedings of the First International Conference on the Simulation of Adaptive Behavior, edited by Meyer J.-A. & Wilson S.W., MIT Press/Bradford Books, 1991.

[8] Meyer J.-A. & Guillot A., Simulation of Adaptive Behavior in Animats: Review and Prospects, In: From Animals to Animats, Proceedings of the First International Conference on the Simulation of Adaptive Behavior, edited by Meyer J.-A. & Wilson S.-W., MIT Press/Bradford Books 1991.

[9] Maes, P., Modeling Adaptive Autonomous Agents. Artificial Life Journal, C. Langton, ed., Vol. 1, No. 1 & 2, MIT Press, 1994.

[10] Bradshaw, J., Software Agents, AAAI Press/The MIT Press, 1997.

[11] Norman, D. The Psychology of Everyday Things, Basic Books, Inc. 1988

[12] Shneiderman, B. Designing the User Interface: Strategies for Effective Human-Computer Interaction -- 2nd Edition, Addison-Wesley 1992

[13] Shiffman, H. Boosting Java Performance: Native Code and JIT compilers http://www.disordered.org/Java-JIT.html 1999

[14] Galyon, E. C++ vs Java Performance http://www.cs.colostate.edu/~cs154/PerfComp/index.html 1998

[15] Pinali , O The Java Performance Report http://www.javalobby.org/features/jpr/ 1999

[16] Sun Microsystems. The Java Management Extensions (Draft 2.0) 1999 http://java.sun.com/products/JavaManagement/

[17] Sun Microsystems. The Java Dynamic Management Kit 3.2 (Programming Guide) 1999 http://www.sun.com/software/java-dynamic/

[18] Almesberger, W. LILO Generic boot Loader for Linux (User's guide & Technical Overview). December 1998

[19] Tierney, B., W. Johnston, B. Crowley, G. Hoo, C. Brooks, D. Gunter. The NetLogger Methodology for High Performance Distributed Systems Performance Analysis, Proceeding of IEEE High Performance Distributed Computing conference (HPDC-7), July 1998

JAVIER JAEN MARTINEZ received his *Diplome d'Etudes Approfondies (DEA)* in Computer Engineering (1st in class rank) in 1995 from the *Institut National des Sciences Appliquees de Lyon* (FRANCE), and his Master's degree in Computer Science in 1998 from the *Virginia Polytechnic Institute and State University* (USA). He is recipient of the *Fulbright* Scholarship and member of the National Computer Science Honor Society *Upsilon Pi Epsilon*. In 1998 he joined the department of computer science at the Polytechnic University of Valencia (Spain) as an assistant professor. Since April 1999 he is a staff member at the Information Technology Division of CERN (the European Laboratory for Particle Physics). His research focuses on agent technology for developing flexible and dynamic middleware layers in scalable clusters for High Energy Physics Computing.