Monitoring System for Distributed Java Applications

Marian Bubak[1,2], Włodzimierz Funika[1],
Piotr Mętel[1], Rafał Orłowski[1],
and Roland Wismüller[3]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
[3] LRR-TUM – Technische Universität München, D-80290 München, Germany
{bubak,funika}@uci.agh.edu.pl, {metel,witch}@icslab.agh.edu.pl, wismuell@in.tum.de
phone: (+48 12) 617 39 64, fax: (+48 12) 633 80 54, phone: (+49 89) 289 28243

Abstract. The paper presents a concept of an implementation of an extension to the On-Line Monitoring Interface Specification for Java Applications. The extension aims at defining an open interface for providing on-line software development tools. The general-purpose, portable, and extensible approach to handle comprehensive monitoring information from a Java run-time environment is intended to span the existing gap between the needs in Java application development tools and the lack of a uniform environment which provides monitoring support for different kinds of tools, like debuggers or performance analysers. The main goal of the resulting monitoring system is to adapt existing tools and build new ones for monitoring Java distributed applications.

Keywords: Java, monitoring system, monitoring interface, distributed object system.

1 Introduction

As a result of its platform independence, Java has become a wide-spread programming language for distributed applications in heterogeneous environments. As Java applications are getting larger and more and more complex, a lack of programming tools that allow to examine and eventually control the behaviour of these applications is becoming apparent. This is especially true for distributed programming. Although there are some tools for profiling distributed Java applications, like JProf [1] or JaViz [2], tools allowing to observe an application's behaviour in more detail or even to control it, are rare. Java debuggers, for example, usually do not support applications distributed across multiple Java virtual machines (JVMs). A notable exception is JBuilder [3]. Furthermore, the tools are incompatible to each other, i.e. they cannot be used at the same time in order to observe different aspects of a program's execution, like e.g. the high-level communication behaviour on the one hand and the detailed execution behaviour of single threads on the other.

Let us have a closer look at *monitoring tools* in general to examine the reasons for this situation. In this context, the term *monitoring* comprises techniques and mechanisms to observe and potentially manipulate applications. Tools that monitor distributed applications have to consider all elements of the distributed system and must control the whole application distributed over different machines. Monitoring tools can be classified into two categories *on-line* and *off-line* tools. The on-line tools run concurrently with the application, thus a user can interactively observe and influence the state of the application. In the case of off-line tools, information on application execution is being stored on disk as a trace file. A tool subsequently uses the data gathered in the trace file for analyses activities.

For observing and possibly manipulating a program state, on-line tools need a specialised module which is called *monitoring system*. Usually this module is tightly integrated with the tool. But even in the case of Java, it must directly interface with the operating system and in addition depends on the specific implementation of the JVM. As a result, tools are rather complex and are not easily portable to different target platforms. In addition, tools come into conflict with each other when used at the same time to observe the same application. A solution to these problems is to have a clearly separated monitoring system that provides a uniform interface for different kinds of on-line tools. The availability of such a system also greatly facilitates the development of new tools.

However, for distributed Java applications, no suitable monitoring systems exist at the moment. Although there are some approaches (e.g. [4]), they are usually targeted towards resource management or profiling only. Our approach for building a universal monitoring system is based on OMIS [5] and the OCM [6]. The OMIS project defines a standardised interface between the tools and the monitoring system. OMIS is not restricted to a single kind of tools, especially it supports tools oriented towards debugging as well as performance analysis and resource management. Originally OMIS has been designed to support parallel systems, and its referential implementation OCM is applied to PVM.

At the beginning the paper outlines the concepts of the extended version of the OMIS specification that supports Java applications. Next we present the architecture of the monitoring system compliant to the specification, the mechanism of interaction with the Java virtual machine, and a sample scenario of interactions between a tool and the monitoring system. The finishing part of the paper focuses on implementation concepts.

2 Interface specification

The specification defines an interface between tool and monitoring system. The main concept of the interface is to provide a special language that allows tools to build complex requests by combining primitive ones in order to achieve the needed flexibility. The individual monitoring functions available by requesting them with an expression in this language are called *services*.

The services relate to the defined elements of monitored applications called *system objects*. The set of system objects creates a view of the monitored application that is "visible" by the tool. This view can dynamically change during run-time, according to the application behaviour or on demand from the tool. The specification distinguishes between two kinds system objects: **execution** objects, i.e. *nodes, JVMs, threads* and **application** objects, i.e. *interfaces, classes, objects, methods*.

The interface defines a set of services for each object which provide information about the object (e.g. about a state of a given thread) – *information services*, allow to manipulate the object (e.g. exchange Java class file during program execution) – *manipulation services* and finally services that trigger arbitrary actions whenever a matching event takes place (e.g. when a given method is entered) – *event services*.

Another important feature of the on-line monitoring interface is the definition of a mechanism that reflects relations between objects of the running application, which is called *conversion*. The following example of a request expressed in the language defined by the interface illustrates this feature:

`:method_get_info([class_id], 1)`

This request calls a method information service and results in obtaining information on all methods implemented in the specified class. The class identifier *class_id* is expanded by the conversion mechanism into a set of method objects. The last parameter determines what kind of information is returned by the service.

This short description only presents the basic ideas behind the interface specification, which are important to understand the realization concepts of the monitoring system. More detailed information about the interface specification can be found in [7] and [8].

3 Architecture of the monitoring system

The specification tries to set up a monitoring interface which is as far as possible independent from the Java virtual machine [9], and from the programming library used for applications. However, any instance of the monitoring system must be implemented for the fixed target architecture and use the fixed access interface to the JVM. As the interface specification tries to be abstract, our goal is to identify a generic monitoring system which can easily be ported among different combinations of software and hardware architectures.

Cooperation between tools and the monitoring system is specified to be based on service requests and replies to them. A service request is sent to the monitoring system as a coded string which describes which activities have to be invoked on the occurrence of a specified event. As a matter of fact, service requests program the monitoring system to listen to event occurrences, perform actions, and transfer results back to the tool. Thus, the monitoring system follows an event/action working model. Cooperation with tools is also covered by this model as the receipt of service requests represents an event to which an appropriate action belongs.

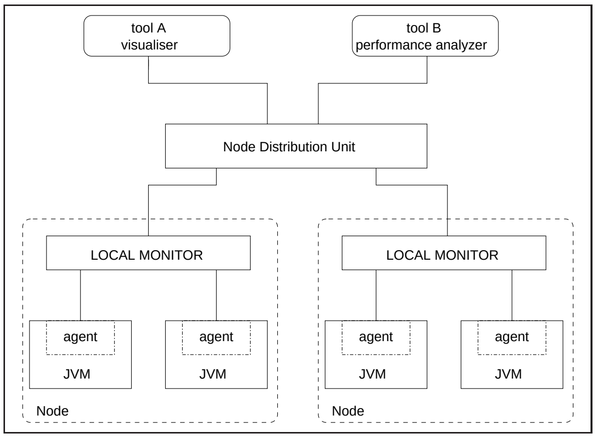

Fig. 1. Architecture of the monitoring environment

Fig. 1 shows the software module structure of the monitoring system. On the top of the Fig. 1 there are various tools running e.g. on a remote workstation, which communicate with the central component of the monitoring system. This component is called *Node Distribution Unit* (NDU).

Node Distribution Unit has to analyse each request issued by a tool and must split it into pieces that can be processed locally by the local monitors on the nodes involved. E.g. a tool may issue the request

`:jvm_run_gc([j_1,j_2])`

in order to run the garbage collector on the JVMs identified by the tokens `j_1` and `j_2`. In this case, the NDU must determine the node executing JVM process `j_1` and the node executing `j_2`. If the JVMs are located on different nodes, the request must be split into two separate requests, i.e. `:jvm_run_gc([j_1])` and `:jvm_run_gc([j_2])`, which are sent to the proper nodes. The NDU must also assemble the partial answers which are received from the local monitor processes into a global reply sent to the tool. Addition and removal of nodes is also detected and handled by the NDU.

The distributed part of the monitoring system consists of multiple *Node Local Monitors* (NLM), one per node of the target system and *Java Virtual Machine Local Monitors* (JVMLM), one per process of the JVM (which can be multiple on one node).

Node Local Monitor processes control the JVM processes via agents (JVMLMs) and operating system interfaces. The NLM offers a server interface that is similar to the monitoring interface, with the exception that it only accepts requests that can be handled locally. This approach can simplify the implementation of the local monitor, without paying any attention to the fact that it is part of a distributed monitoring system. The NLM is responsible for cooperation with JVMLMs, e.g. for *start/stop* or *attach/detach* to local JVMs. In addition, it gathers information from the outside of the JVM process i.e. about the operating system and the hardware.

Java Virtual Machine Local Monitor is the *agent* that is embedded in the virtual machine process. The agent is responsible for execution of the requests received from the NLM. Its implementation depends on the virtual machine native interfaces that provide low level mechanisms for interactive monitoring of the Java virtual machine.

In order to achieve independence of the monitoring system from a concrete Java virtual machine implementation, it is required to build JVMLMs that will use standard JVM interfaces like JVMPI [10], JNI [11], JVMDI [12] or Java bytecode instrumentation. In the next section we consider JVMPI as a possible basis for the implementation of JVMLM.

4 Interaction with JVM

Low level interactions with the Java virtual machine are a critical part of the monitoring system. The component responsible for this activity (JVMLM) has to provide functionality that allows to realize services defined in the interface specification. One of the possible approaches is using the JVMPI interface to access the JVM.

JVMPI defines a mechanism for obtaining profiling data from the Java virtual machine. Profiling data provides information about events that occur during application run time, in the form that enables to track the cost of these events, e.g. it can show what portion of the program allocates the greatest amount of memory. Building the JVMLM on top of JVMPI is possible due to the following features of the interface:

- JVMPI is an interface between JVM and a *profiler agent* that runs in the same process. The agent is implemented as a dynamic library which is loaded by JVM at startup.
- JVMPI defines a set of events that can be sent to the agent by JVM during Java program execution. JVM notifies the agent about such activities like: method *enter/exit*, class *load/unload*. These events are a low level realization of some event services from the monitoring interface specification.
- JVMPI provides a set of callback functions implemented by the Java virtual machine that allow the agent to obtain additional information in response to an event notification and for basic threads and garbage collector manipulations, like: thread *suspend/resume/get backtrace*, garbage collector *enable/disable/run*.

Thread start event report
described by: thread object id, thread id, thread name, thread group

```
THREAD START (obj=814dd10, id = 4,  name="Finalizer",  group="system")
THREAD START (obj=822de00, id = 6,  name="Producer#0", group="main")
THREAD START (obj=82302a0, id = 7,  name="Consumer#0", group="main")
```

Thread stack trace *consists of: trace number, thread id, set of frames.*
Frame *consists of: class name, method name, source file name, line number.*

```
TRACE 15: (thread=4)
   java/lang/Object.wait(Object.java:Native method)
   java/lang/ref/ReferenceQueue.remove(ReferenceQueue.java:112)
   java/lang/ref/ReferenceQueue.remove(ReferenceQueue.java:127)
   java/lang/ref/Finalizer$FinalizerThread.run(Finalizer.java:174)
...
MONITOR DUMP BEGIN
    THREAD 8,  trace 9,   status: MW
    THREAD 7,  trace 8,   status: MW
    THREAD 6,  trace 14,  status: CW
    THREAD 4,  trace 15,  status: CW
    THREAD 2,  trace 17,  status: R
    ...
    MONITOR Buffer(8230258)
        owner: thread 6, entry count: 1
        waiting to enter: thread 7, thread 8
    MONITOR java/lang/ref/ReferenceQueue$Lock(814f4e8) unowned
        waiting to be notified: thread 4
    ...
MONITOR DUMP END
```

Thread status:
MW - *waiting on a monitor*
CW - *waiting on a condition variable*
R - *runnable*
trace number refers to current thread stack trace

Monitor state *described by: name, object id which is associated with a monitor, monitor owner thread, entry to monitor count, waiting threads.*

Fig. 2. JVMPI agent reports.

In order to show what kind of information can be obtained via the JVMPI interface we present in Fig. 2 parts of the reports generated by a simple agent during a program execution that simulates a producer - consumer problem.

The first report provides information about a thread start event generated by the JVM. The next one shows an example thread stack trace (for the "Finalizer" thread). This is a result of a JVMPI call-back function called by the agent. Very informative is the report of monitors dump. It shows the statuses of threads and states of the Java synchronization monitors.

JVMPI provides only low level information about physical aspects of a running application, but it offers enough data to build higher level services defined in our monitoring interface specification. The whole picture of requests flow in the monitoring system is shown in the next section.

5 Monitoring scenario

In order to outline the interaction between layers in the monitoring system and to show how the monitoring interface can by used by a tool, we describe a monitoring scenario for an *instrumentation tool*.

The interface specification defines services that allow to instrument Java classes. Instrumentation enables to insert additional bytecode into compiled Java methods that can be executed. This bytecode can realize some monitoring system functionality, for instance, the timestamps of the method call measurements. Fig. 3 shows a possible architecture of the monitoring system with the code instrumentation tool. The instrumentation tool sends requests to the monitoring system in order to exchange a Java class file before it is loaded by the JVM.

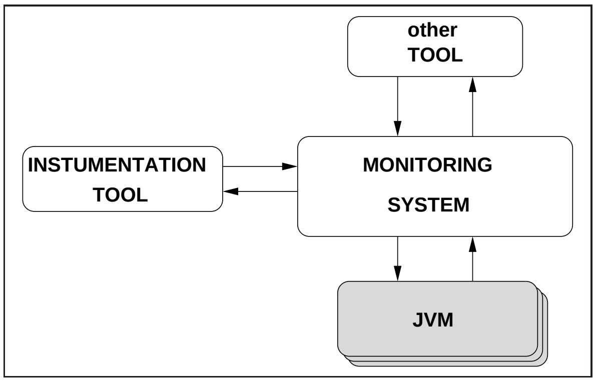

Fig. 3. Instrumentation tool

Let's assume the tool is interested in instrumenting the *java.lang.Thread* class. Then, the following request can be used:

```
jvm_loaded_hook([j_13],"java.lang.Thread"):
                            class_exchange([$class],newData)
```

This is a conditional [1] request. The monitoring system executes the *class_exchange* action after the event *jvm_loaded_hook* occurs. The parameters for the event service determine which JVM is monitored (`j_13`) and which Java class should be considered. Parameter `$class` in the manipulation service indicates the class hook returned by the event. `newData` is a reference to the instrumented class file.

An additional request can be sent by a tool to obtain notification that the class has been correctly loaded by the JVM.

[1] A conditional request consists of an event definition and a list of actions that is executed whenever a matching event occurs.

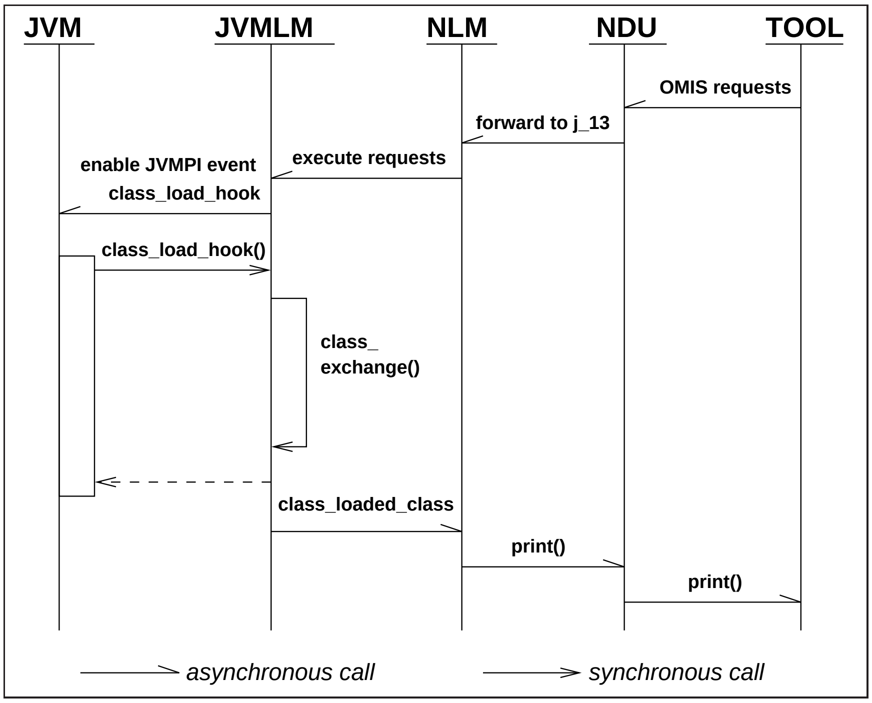

Fig. 4. Sequence diagram of the request execution

```
class_loaded_class([c_java/lang/Thread_j_13],[],[]):
                          print(["Class",$class,"loaded.OK."])
```

The parameter `c_java/lang/Thread_j_13` is a monitoring system representation of class *java.lang.Thread* on JVM with id `j_13`.

Fig. 4 presents a sequence diagram that shows the stages of execution of these two requests that are input data for the monitoring system. On the output, the message is returned to the tool, but the real effect of those requests is loading the instrumented class file by the JVM. There are only shown the basic components of the monitoring system that are involved in the following scenario:

1. First, the tool sends asynchronously the requests to the monitoring system. In order to minimize communication overhead, these two requests can be sent in one stream.
2. The central part of the monitoring system (NDU) parses the requests, and – if it is required – splits them and then sends them to a proper NLM. In this example, requests are sent to the one NLM placed on the node where the JVM with id `j_13` can be found.
3. NLM instructs the appropriate JVMLM how it should react on the event of *java.lang.Thread* class loading.

4. JVMLM starts to filter JVM events sent via the JVMPI interface about class loading. When the JVMPI event (`class_load_hook()`) occurs for class *java.lang.Thread*, then JVMLM can exchange the class file, so JVM loads the instrumented one. All these operations are done in the same JVM thread which loads the classes. Thus, in this case there is no need to stop the JVM, but for more complicated actions, it may be required to stop some of the JVM threads, and send the notification about an event to the NLM which also may forward the event to the NDU in order to coordinate the actions with other JVMs (on other nodes).
5. *class_loaded_class* is a monitoring system event generated by JVMLM after successful *java.lang.Thread* class instrumentation. On occurrence of this event, a short notification is sent back to the tool.

The analysis of the sequence of requests shows that the request processing is distributed over all components and wherever only possible, communication between them is asynchronous. Since NDU and NLM coordinate request processing and activities between different Java virtual machines, it is possible to implement a mechanism for tracking remote methods calls. Such a mechanism is defined in the extension to the OMIS specification and an example scenario that presents its realization can be found in [8].

6 Concluding remarks

In the paper we presented a model of a monitoring system underlying the functionality of tools used for distributed Java applications. The interface between the tools and the monitoring system is based on an extension to the OMIS specification. The structure of the environment comprises a central component for handling tool requests and replies to them and a set of distributed components for control of JVM processes via profiler agents provided within the JVMPI interface.

The low-level interaction with JVM is intended to supply indispensable information on event occurrences the tools may be interested in during the execution of the program as well as triggering proper actions, based on callback functions provided within JVM.

A monitoring scenario was presented to illustrate a feasible interaction within the monitoring system infrastructure, to reveal the need in establishing a well-defined communication and coordination mechanism between the layers of the monitoring system.

The approach presented can be used for building uniform tool environments for Java distributed applications, independent of the peculiarities of particular platforms and JVMs.

References

1. G. Pennington and R. Watson: JProf – a JVMPI Based Profiler `http://starship.python.net/crew/garyp/jProf.html`

2. I.H. Kazi, D.P. Jose, B. Ben-Hamida, C.J. Hescott, C. Kwok, J. Konstan, D.J. Lilja, and P.-C. Yew: JaViz: A Client/Server Java Profiling Tool, *IBM Systems Journal*, **39**(1), (2000) 96-117.
`http://www.research.ibm.com/journal/sj/391/kazi.html`
3. Borland: JBuilder 3.5 Datasheet
`http://www.borland.com/jbuilder/jb35/datasheet.html`
4. P. Bellavista, A. Corradi and C. Stefanelli: Java-based On-line Monitoring of Heterogeneous Resources and Systems. In 7th Workshop HP OpenView University Association, Santorini, Greece, June 2000.
`http://lia.deis.unibo.it/Staff/PaoloBellavista/papers/hpovua00.pdf`
5. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode: OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series, (1997)
`http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz`
6. R. Wismüller, J. Trinitis and T. Ludwig: A Universal Infrastructure for the Runtime Monitoring of Parallel and Distributed Applications. In Euro-Par'98, Parallel Processing, volume 1470 of Lecture Notes in Computer Science, pages 173-180, Southampton, UK, September 1998. Springer-Verlag.
7. M. Bubak, W. Funika, P. Metel, R. Orlowski, and R. Wismüller: Towards a Monitoring Interface Specification for Distributed Java Applications. In Proc. 4th Int. Conf. on Parallel Processing and Applied Mathematics - PPAM 2001, Naleczow, Poland, September 2001. To appear.
8. P. Mętel and R. Orłowski: A Java-oriented extension to On-line Monitoring Interface Specification. Master Thesis. Institute of Computer Science, Stanislaw Staszic University of Mining and Metallurgy, Kraków, 2001.
9. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
10. Sun Microsystems: Java Virtual Machine Profiler Interface (JVMPI)
`http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html`
11. Sun Microsystems: The Java Native Interface Specification
`http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html`
12. Sun Microsystems: Java Virtual Machine Debug Interface (JVMDI)
`http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/jvmdi.html`