A MONITORING PLATFORM FOR DISTRIBUTED JAVA APPLICATIONS

WŁODZIMIERZ FUNIKA[1], MARIAN BUBAK[1,2],
MARCIN SMĘTEK[1] AND ROLAND WISMÜLLER[3]

[1] *Institute of Computer Science, AGH University of Science and Technology,*
Al. Mickiewicza 30, 30-059 Cracow, Poland
{funika, bubak}@uci.agh.edu.pl, smentos@icslab.agh.edu.pl

[2] *Academic Computer Centre CYFRONET,*
Nawojki 11, 30-950 Cracow, Poland

[3] *Fachgruppe BVS – Universität Siegen,*
Hölderlinstr. 3, D-57068, Siegen, Germany
roland.wismueller@uni-siegen.de

(Received 6 August 2004)

Abstract: This paper presents a new Java oriented monitoring infrastructure that enables tools to observe, analyze and manipulate the execution of distributed Java applications independent of implementation details like instrumentation of monitored entities, hardware platform and application libraries. Tools can access the monitored application via a standardized interface defined by an On-Line Monitoring Interface Specification (OMIS) and extended by a set of new Java-specific services relating to garbage collection, class loading, remote method invocation, *etc.* The new monitoring functionality can be applied for building various kinds of tools and for adapting the already existing ones, such as performance analyzers, debuggers, *etc.*, working in the on-line mode.

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

1. Introduction

Java, as a relatively simple, object-oriented, secure and portable language, is also a flexible and powerful programming system for distributed computing. Program development with Java results in software that is portable across multiple machine architectures and operating systems. Distributed programming in Java is supported by remote method invocation (RMI), object serialization, reflection, a Java security manager and distributed garbage collection. Java RMI is designed to simplify the communication between objects in different virtual machines allowing transparent calls to methods in remote virtual machines.

A major disadvantage of Java is the speed of execution, especially in the case of distributed applications, due to the limited bandwidth of the communication channel and the added delay caused by the JVM translating the byte code and

garbage collection. A developer of distributed systems faces various problems that make developing such systems more difficult that expected. In a complex distributed application, performance optimization of the code becomes much more important and requires programmers' close attention. Understanding the nature of Java-related problems allows programmers to build large and scalable applications. However, without a suitable performance analysis tool for Java programs, it is often difficult to analyze a program for performance-tuning and bug detection. Thus, there is a need for tools (performance analyzers, debuggers, *etc.*) that allow programmers to control and improve their applications.

The goal of our research [1–5] is to elaborate a comprehensive tool support for building distributed Java applications by providing uniform, extendable monitoring facilities for communication between components, for analyzing an application's execution to understand system behaviour, and for detecting bugs.

In this paper, we focus on the issues of building a monitoring platform of the above properties based on a well-defined interface between a monitoring system organized as middleware and tools that use the monitoring facilities provided by the monitoring middleware. The paper is organized as follows. A short overview of related work in Section 2 is followed by considerations of Java tools' functionality (Section 3). An overview of OMIS and an OMIS-compliant monitoring system, OCM, and its Java-oriented extension is presented in Section 4. Next, an outline of the Java-oriented monitoring system's architecture is given in Section 5, followed by an analysis of the J-OCM infrastructure as a kind of middleware with its attributes (Section 6). The event model in OMIS and its extension by Java-specific events are presented in Section 7. An overview of the start-up procedure when using J-OCM is described in Section 8. Work on performance analysis tools is briefly outlined in Section 9. Conclusions and future work are summarised in Section 10.

2. Related work

The first version of JVMs had poor support for monitoring Java programs. Initially there was a simple debugger, `jde`, attached to the Java Development Kit (JDK). Then, there was an instrumented Java virtual machine build for JDK version 1.16 to support the collection of profiling data generated when executing a Java program. This approach was developed until version 2 of the Java platform. All JVMs for the new Java platform were equipped with interfaces for debugging (JVMDI) [6] and profiling (JVMPI) [7]. A new release of Java 2 Platform version 1.5, called Tiger, contains a new native profiling interface called JVMTI which is intended to replace JVMPI and JVMDI. JVMTI aims to cover the full range of native in-process tools access, which in addition to profiling, includes monitoring, debugging and, potentially, a wide variety of other code analysis tools. Additionally, Tiger's implementation includes a mechanism for bytecode instrumentation, the Java Programming Language Instrumentation Service (JPLIS). This enables performance analysis tools to execute additional profiling only where needed. The advantage of this technique is that it allows for more focused analysis and limits the interference of the profiling tools in a running JVM. The instrumentation can even be dynamically generated at runtime, as well as at class loading time, and pre-processed as class files.

Most of the tools for JVM versions from 1.2 to 1.4 are based on the Java Virtual Machine Profiling Interface (JVMPI) [7]. Starting with JDK 1.2 SDK it also includes an example profiler agent for efficiency examination called `hprof` [8], which can be used to build professional profilers. A Heap Analysis Tool (`Hat`) [9] enables one to read and analyze profile reports of the heap generated by the `hprof` tool and may be used *e.g.* for debugging "memory leaks". `JTracer` [10] is a debugger which provides traditional features, *e.g.* a variable watcher, breakpoints and line-by-line execution. `J-Sprint` [11] provides information about what parts of a program consume the most of execution time and memory. `JProfiler` [12], targeted at JEE and JSE applications, provides information on CPU and memory usage, thread profiling and VM. Its visualization tool shows the object references chain, execution control flow, thread hierarchy and general information about JVM using special displays. There is also a group of powerful commercial tools with friendly graphical interfaces: `OptimizeIt` [13], `Jtest` [14] and `JProbe` [15, 16], which enable identification of performance bottlenecks.

All these tools have similar features: memory, performance, code coverage analysis, program debugging, thread deadlock detection and class instrumentation, but many of them are designed to observe a single-process Java application and do not support directly monitoring a distributed environment based on RMI middleware, except for `JaViz` [17], which is intended to supplement the existing performance analysis tools with tracing client/server activities to extend Java's profiling support for distributed environments.

The above mentioned tools provide a wide range of advanced functionalities, but practically each of them only provides a subset of the desired functions. Distributed systems are very complex and the best monitoring of such a system could be achieved by using diverse observation/manipulation techniques and mechanisms. It is therefore often desirable to have a suite of specialized tools such as debuggers and performance analyzers, each of them addressing a different support issue and allowing developers to explore the program's behaviour from various viewpoints. Therefore, there is a need to establish a more general approach to build flexible, portable and efficient monitoring-based tools.

3. Building a monitoring system

In order to provide comprehensive support for distributed programming, the tools need to be able to *manipulate* and *observe* the whole application distributed over different machines on-line, in contrast to the *off-line* mode, which assumes collecting information at runtime for later analysis (*e.g.* `JaViz`). To provide flexible functionality of tools, the monitoring activities underlying access to the observed application should be concentrated in a separate module, called the *monitoring system*. The monitoring system should ideally provide a uniform on-line interface for different kinds of tools, which allows to easily build tools without a need to understand the implementation of monitoring.

Most tool environments consolidate tools with a monitoring facility into a single monolithic system. However, there are some approaches which can underlie the construction of particular types of tools. The `Java Platform Debugger Architecture` (`JPDA`) [18] provides an infrastructure to build a debugger tool by allowing access to

the internals of a program running on one JVM from within another. JPDA can be a good platform for remote debugging, but it does not offer support for distributed debugging and monitoring. Moreover, it only allows a single connection to the target JVM at a time, which means that different tools cannot be used to monitor the same JVM simultaneously.

A new release of Java 2 Platform version 1.5 introduces a new JVM Monitoring and Management API, Java Management Extensions (JMX) [19] for Java Virtual Machine which specifies a comprehensive set of JVM internals such as loaded classes and threads, as well as runtime information about the virtual machine (uptime, memory usage, garbage collection statistics, system properties) that can be monitored from a running JVM. It provides information on low memory and monitor deadlock, and details on the operating system on where the JVM is running. This information is accessed through Java Management Extension MBeans and can also be accessed remotely via the JMX remote interface and with industry standard SNMP tools. JMX is a well designed and mature technology that results from research which started in 1995, while the first release of the standard appeared four years ago, JMX is the outgrowth of the pioneering Java Management API (JMAPI) technology (1995) and the later Java Dynamic Management Kit.

4. From OMIS to J-OMIS

The work on building a versatile monitoring system followed the idea of a monitor/tool interface specification, `OMIS` [20], and its implementation, `OCM` [21]. A tool contacts the monitoring system via a standardized interface. The cooperation between the tool and the monitoring system is based on the *service request/reply* mechanism. A tool sends a service request to the monitoring system, *e.g.* a coded string which describes a condition (event) (if any) and activities (action list) which have to be invoked (when the condition becomes true). In this way the tool programs the monitoring system to listen for event occurrences, perform appropriate actions, and transfer results to the tool. OMIS relies on a hierarchy of abstract objects: nodes, processes, threads, messages and message queues. Each object is represented by an abstract identifier (*token*) which can be converted into an other token type by conversion functions *localization* and *expansion*, automatically applied to every service definition that has a token that refer to an object type different from that for which the service is defined. At each moment each tool has a well-defined scope, *i.e.* it can observe and manipulate a specific set of objects *attached* on request from a tool. OCM, the first OMIS-compliant monitoring system, has a distributed architecture: local monitors, one per node (LM), and a Node Distribution Unit (NDU) that has to analyze each request issued by a tool and split it into separate requests that can be processed locally by LMs on the proper nodes.

By allowing the monitoring system to be expanded with a tool or monitor extension, via adding new services and new types of objects to the basic monitoring system, OMIS enables the provision of monitoring support for specific programming environments. *Java-bound OMIS (J-OMIS)* is a monitor extension to OMIS for Java applications intended to support the development of Java distributed applications with on-line tools. J-OMIS introduces a range of new types of objects, new services

and conversion functions. The extension divides a new Java-bound object hierarchy into two kinds of system objects: execution objects (*i.e. nodes, JVMs* and *threads*) and application objects (*i.e. interfaces, classes, objects* and *methods*). J-OMIS defines a set of services operating on each object. As in the original OMIS, J-OMIS specifies three types of services: (i) information services, providing information about an object, (ii) manipulation services, allowing to change the state of an object, and (iii) event services to trigger arbitrary actions whenever a matching event takes place. J-OMIS defines the relations between the objects of a running application, which are expressed by conversion functions, *e.g. implements* or *downcast/upcast*. The idea of conversion comes from OMIS 2.0, where the *localization/expansion* conversion is defined. J-OMIS enlarges this ability by a few additional operations that result from the object-oriented nature of Java, *e.g.* the request `: method_get_info([class_id], 1)` relates to the `method` information service to obtain information on all methods implemented in the specified class. The `class_id` token is expanded by the conversion mechanism into a set of *method* objects [1]. Monitoring of distributed programs based on the RMI mechanism requires the establishment of an additional set of services. They include services relating to the RMI registry: `rmiregistry_get_info()` to obtain the names of objects registered on a given machine and `rmi_registry_object_get_info()`, to get the names of all methods of a given remote object. The RMI-related vent services refer to the progress of remote method invocation in the client: *start/end of an RMI call*, *sending RMI call to/receiving results from* the server, and in the server: *start/end of handling* the request, *start/end of calling the method (locally)* of a remote object.

5. The architecture of the Java-oriented monitoring system

Based on J-OMIS, a Java-oriented extension to OCM, J-OCM has been designed through extending the functionality of OCM, adding new software components and adapting the existing ones. This approach allows one to combine the existing functionality of OCM with the Java platform to support Java homogeneous and heterogeneous computing in the future. Figure 1 shows which components have been added or modified. The *Node Distribution Unit* (NDU) is an unchanged part of the whole monitoring infrastructure, which is still responsible for distributing requests and assembling replies; *e.g.* a tool may issue a request in order to run the Garbage Collector on specified JVMs. Therefore NDU must determine the nodes executing the JVMs and, if needed, to split the request into separate sub-requests to be sent to the proper nodes. NDU's aim is to make the whole system manageable, thus it has to program the local monitors of all the currently observed nodes. As the set of monitored nodes may change over time, NDU must properly react to these changes: to create local monitors on newly added nodes or to re-arrange its list of objects involved in the execution of requests that have been issued by tools when some nodes are deleted.

The *Local Monitor* is a monitor process, independent from the whole global monitoring infrastructure. Each monitor process provides an interface similar to that of NDU, with the exception that it only accepts requests to operate on local objects.

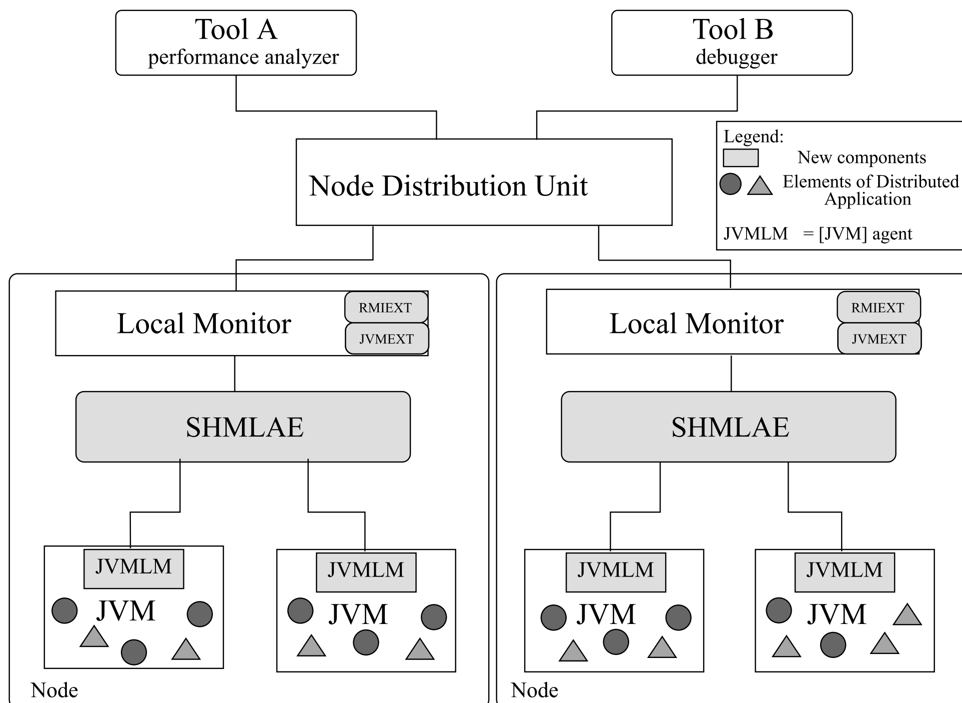

Figure 1. J-OCM architecture

To support the monitoring of Java applications, LM's extension, JVMEXT, provides new services defined by J-OMIS, which control JVM via agents. LM stores information about the target Java application's objects, such as *JVMs, threads, classes, interfaces, objects, methods*, *etc.*, referred to by tokens. Another monitor extension, RMIEXT, is intended to notify J-OCM of RMI-related events and obtain information from the RMI registry.

The most essential aspect of building J-OCM is the way in which monitoring data about the application and JVM can be handled and delivered to the monitoring system. JVM has the ability to load an additional native code in the form of a shared library into its address space. We have used this feature and implemented the *Java Virtual Machine Local Monitor*, an agent embedded into a JVM process (Figure 2) which is responsible for the execution of requests received from LM. It provides the same interface as the monitoring extensions to LM, implemented by its local representations of JVMEXT and RMIEXT which operate on the objects local to a given JVM. JVMLM uses JVM native interfaces such as JVMPI, JVMDI, JNI to access low-level mechanisms for interactive monitoring of JVM.

The monitoring of RMI requires additional instrumentation based on modifications to Java RMI classes, `sun.rmi.server.UnicastRef` and `sun.rmi.server.UnicastServerRef`, to handle remote calls on the client and the server side, respectively. An additional code notifies JVMLM of the path and time of RMI calls in particular phases via the JNI mechanism [22].

The *Shared Memory-based Local Agent Environment* (SHMLAE) is a communication layer to support cooperation between agents and LM. It offers *non-blocking*

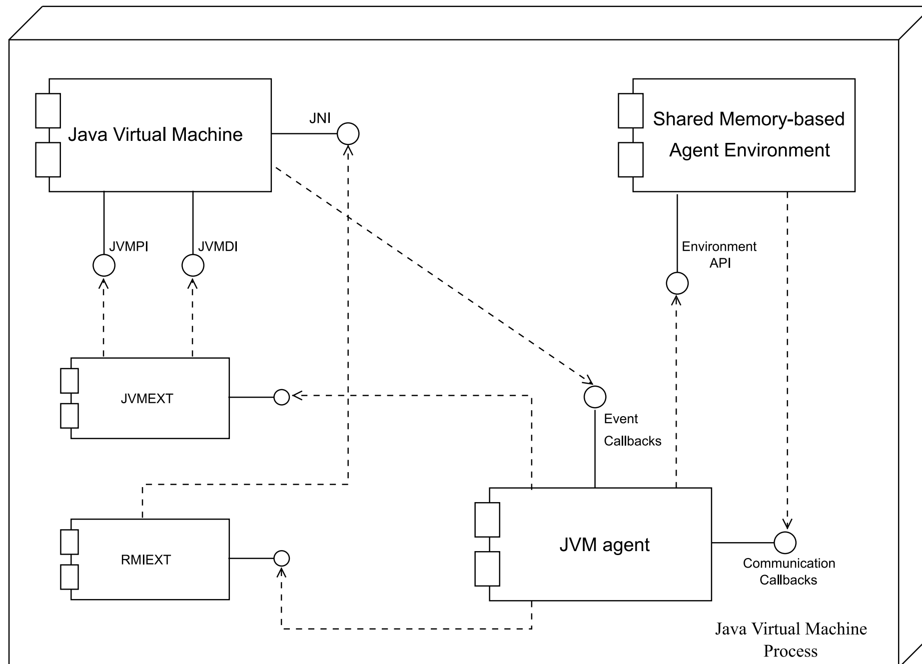

Figure 2. JVM agent

send and *interrupt-driven receive* operations to support monitoring techniques used in OCM, based on the event-action paradigm.

6. J-OCM as middleware

Every distributed environment, such as CORBA, COM/DCOM or Java RMI, provides mechanisms to simplify the process of developing a distributed application. Similarly, the leading idea of building a Java-bound monitoring system is to provide a system framework that allows new services to be added as monitoring extensions (*e.g.* in the form of plugins for programming techniques like RMI, CORBA and Web Services). J-OCM is more than a single distributed application; it provides a number of utilities which can help tool developers extend the functionality of Local Monitors and JVM agents (see Figure 3). In this way, the whole monitored system can be seen as a set of distributed objects and the monitoring system – as a higher-level software layer, *middleware*, that provides a standardized interface for the tool to access the monitored objects, regardless of implementation details.

A distributed client-server system is based on the principle that the definition of behavior (a *client's* concern) is separated from the behavior's implementation (a *server's* concern). To meet the requirements of distributed computing, a distributed system has to comprise additional architectural elements: an object interface specification, an object stub and skeleton, an object manager, a registration/naming service and a communication protocol. To deal with the distributed target system, we shall consider the functioning of J-OCM as a distributed client-server system, focusing on the functionality of its components.

Interface definition. The first stage of the process of developing a distributed application is to define the interface of a remote object (*e.g.* methods, data types), written in an Interface Definition Language. Similarly, the leading idea of OMIS is to provide support for building tools and monitoring systems for new environments by extending its functionality. For each extension, an IDL file, called *registry*, has to be provided, that specifies new objects, new services and their attributes.

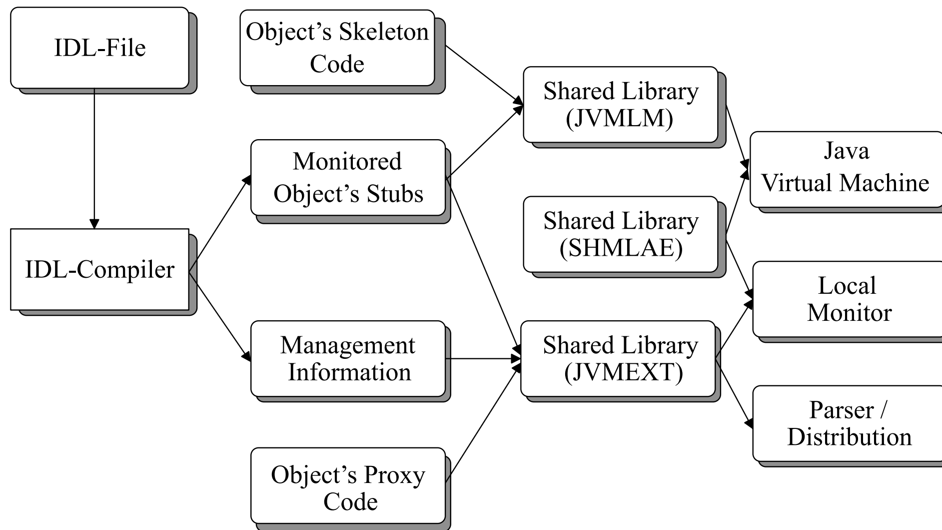

Figure 3. The development process of J-OCM

The stub and the skeleton enable transparent communication between the client and the remote object. They are based on the `Proxy Design Pattern` [23], wherein an object is represented by another object (a proxy) in order to control access to the object. In distributed computing, the role of the proxy is played by the stub, which allows making a local call on a remote object. The skeleton on the server side receives an incoming call and invokes it on the real object. The Proxy pattern is used by J-OCM to provide the tool with transparent access to the monitored objects. The monitored objects are identified by tokens which refer to the proxy object. The proxy is a representation of the real object in the monitoring system. The object's proxy contains all information that is needed to deliver the tool's requests to the JVM agent (JVMLM) which directly accesses JVM. The JVM agent acts as a skeleton, while the remote proxy which is embedded into JVM is a platform-dependent native library. The agent transforms a call and its parameters received from LM into the format required by one of the interfaces used to interact with JVM.

The object manager and the registration/naming service. Remote-method calls issued by the client are routed through the object manager to the proper object on the server. The object manager also routes the results back to the client. The *registration/naming service* acts as an intermediary layer between the object client and the object manager. Once an interface to the object is defined, an implementation of the interface needs to be registered with the naming service so that the object can be accessed by clients using the object's name. The main components of the J-OCM – NDU

and LMs – can be classified as object managers and provide operations similar to the naming service in distributed systems. Any object that can be observed/manipulated by tools is represented by a *token*, which is a globally unique name of the monitored object.

The tokens and proxies of the relevant monitored objects are created when: (1) the JVMLM is started and notifies the node's LM of its existence (*e.g.* `jvm_j_1`), (2) events referring to the creation of *threads*, *classes*, *objects*, *interfaces*, *etc.* are raised and the tool is interested in them, or (3) the tool issues information requests of the syntax: {`jvm, thread, class,` *etc.*}`_get_tokens()` to obtain a list of tokens of all the monitored objects of a given class.

7. Handling of events

In an event-based monitoring system, basic events are captured by sensors which are inserted into target systems and notify the monitoring system whenever an event occurs. The monitoring system takes some action(s) associated with the event. These actions can either carry out data collection or manipulate the running program. In J-OCM, event notification is supported both by LMs and JVMLMs.

Java-oriented techniques for event-based monitoring. JVM notifies JVMLM about several internal events, using JVMPI and JVMDI. These events are fired by changes in the state of Java threads like (*started*, *ended*, *blocked on a locked monitor*), the beginning/ending of an invoked method, class loading operations, object allocation/de-allocation, and the beginning/ending of garbage collection, exception throwing, *etc.*

To support interactive observation of the target system, all events must be processed by the JVM agent, while the agent sends the events to LM selectively, to avoid too much overhead on LM. This is based on a filtering mechanism introduced into the JVM agent to select events that should be sent to LM. To control the forwarding of events, the agent uses a filter in the form of a table, in which it stores information about what events LM is interested in. LM can *stop* or *resume* notification of specific events sent from its agents.

The **OMIS event model**, along with the *event* term, additionally defines the *event class* predicate, specifying a set of event occurrences, a pattern defining the events of interest. In OCM, event classes are represented by an event service with its parameters. The event processing in OCM is based on the idea that event classes form a tree hierarchy, where the root is the universal class containing all the detectable events. New event classes are derived from the existing ones with filters that allow only certain events to pass. Each class in the event class tree may be associated with a set of filters, each of which is used to derive a more specific event class from a more general one. In addition, some event classes may have their associated action lists. When an event is detected by OCM, it is matched against all the event classes in the tree by performing a tree traversal starting with the tree's root which matches all events. At each traversed node, the filters associated with the node are evaluated. If a filter is evaluated to true, the event matches the derived event class, which is therefore traversed as well. During this tree traversal, the action lists associated with the event classes matched by the event are scheduled for execution.

Event-based interaction. Interactions between the Local Monitor and its Java agents (JVMLMs) are the most critical part of the J-OCM event system. The Java agents use SHMLAE to notify LM about event occurrence and to transfer event-specific parameters. Before being handled by LM, an event must be enabled in the specific JVMLM. This is done when a tool enables the previously defined event service request by issuing a `csr_enable` service. Once the JVM agent has received the request from LM, it starts passing events to LM which must take care of handling them.

OCM uses signals as asynchronous mechanism to notify about message arrival and defines a special event class, called `mon_received_signal(integer sig_no)`, triggered when the monitor process receives the specified signal. The use of this event class is a solution to handle events from JVM agents and then process them according to the OMIS event model. The root of a Java-specific event tree derives from the `mon_receive_signal` event class. Events generated by JVMs are captured whenever a proper Local Monitor gets a signal sent by an agent. Then LM uses a non-blocking receive call to receive the messages containing information about an event that has occurred in a monitored JVM and process it according to the OMIS event model.

8. Start-up procedure

To start monitoring a Java application, we should bear in mind a number of issues. *First*, it is necessary to replace the original standard Java core classes (known as bootstrap classes) with the instrumented ones provided by J-OCM by passing the path to the modified classes at the start-up of JVM. This can be accomplished with the `-Xbootclasspath` option, which permits to specify an alternative location for a code which, for instance, overrides the standard RMI libraries.

The path to the modified classes must be placed at the beginning of the lists of directories where the classes reside. This can be achieved by adding `/p` to the above option. The *next* change to the start-up of JVM is an additional option, `-Xdebug`, necessary when debugging applications within JVM. It is needed when the JVMDI interface is used. The *third* change is loading the agent to JVM, which can be done with the `-Xrunjvmlm` option, which loads and initializes agents. The start-up of an exemplary RMI application with enabled RMI monitoring may look like this:

```
java -Xbootclasspath/p:$INSTRUMENTED_CLASS_DIR -Xdebug
-Xrunjvmlm RMIExampleServer
java -Xbootclasspath/p:$INSTRUMENTED_CLASS_DIR -Xdebug
-Xrunjvmlm RMIExampleClient
```

The `$INSTRUMENTED_CLASS_DIR` environment variable expands to the directory where the instrumented classes' package is located. Once the above commands are performed, monitoring agents (JVMLM) will register the launched Java virtual machines in their LM to make them visible for tools. After attaching to the node where the launched JVMs are running the tools are able to get tokens (references) to these virtual machines. When the tokens are obtained, the tools can perform any of the services defined in J-OMIS.

9. Performance tool

Once the needed functionality of J-OCM had been achieved, we started the design and implementation work on performance analysis tools. One direction was connected with on-line profiling of Java applications, while another aimed at monitoring RMI calls, especially with respect to their progress.

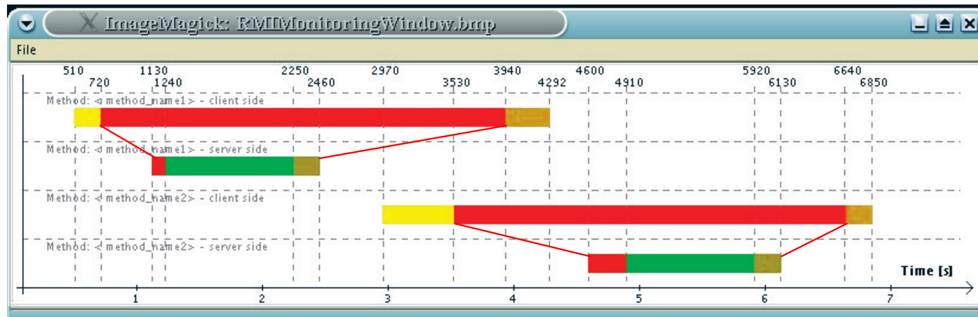

Figure 4. A prototype OMIS-based tool's diagram for monitoring remote-method calls

An example of a RMI monitoring session is shown in Figure 4. The tool uses raw monitoring data supplied by J-OCM to visualize interactions between clients and servers when performing RMI calls with a space-time diagram. The captured RMI-bound events are mapped into relevant time intervals, which enable observation of time spent in the execution phases of RMI.

10. Conclusions

The work on building Java-oriented tools followed the idea of separating the layer of tools from a monitoring system's functionality. We extended the On-line Monitoring Interface Specification by a Java-specific hierarchy of objects and a set of relevant services. The work on a Java-oriented monitoring system, J-OCM, was focused on extending the functionality of Local Monitors, which are the distributed part of the system, and introducing new software levels interfacing J-OCM to JVM and providing a communication mechanism for the low-level layers.

To deal with the target Java system we considered it in terms of the distributed system architecture, which allowed to separate the work on the definition of services on the tool side from their implementation on the server side provided by the monitoring system. We extended the original event model provided in OCM by a Java-specific event sub-model which covers the functioning of the basic application- and execution-related entities of the Java distributed application. In order to enable the monitoring of RMI calls with the J-OCM system, we used the possibility to replace classes, characteristic for the Java platform. These changes have enabled notification of events which reflect the course of an RMI call, as well as obtaining information needed for the implementation of information services.

Our on-going work is focused on completing the implementation work on J-OCM and, at the same time, on implementing a set of Java-oriented tools for performance analysis. Our future plans are connected with moving to a new monitoring mechanism

in a new release of the Java platform and building J-OMIS-based debugger for distributed Java applications.

Acknowledgements

This research was carried out as a Polish-German collaboration project and was partially supported with KBN grant no. 4 T11C 032 23.

References

[1] Bubak M, Funika W, Mętel P, Orłowski R and Wismüller R 2002 *Proc. 4th Int. Conf. PPAM 2001*, Naleczow, Poland, LNCS **2328** 315
[2] Bubak M, Funika W, Smętek M, Kiliański Z and Wismüller R 2003 *Proc. 10th European PVM/MPI Users' Group Meeting*, Venice, Italy, LNCS **2840** 447
[3] Bubak M, Funika W, Wismüller R, Mętel P and Orłowski R 2003 *Future Generation Computer Systems* **19** 651
[4] Bubak M, Funika W, Smętek M, Kiliański Z and Wismüller R 2004 *Proc. 5th Int. Conf. PPAM*, Czestochowa, Poland, LNCS **3019** 352
[5] Funika W, Bubak M, Smętek M and Wismüller R 2004 *Proc. Int. Conf. on Computational Science*, Cracow, Poland, LNCS **3038** 472
[6] Sun Microsystems: Java Virtual Machine Profiler Interface (JVMDI), http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jvmdi-spec.html
[7] Sun Microsystems: Java Virtual Machine Profiler Interface (JVMPI), http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html
[8] The SDK Profiler, http://www.javaworld.com/javaworld/jw-12-2001/jw-1207-hprof.html
[9] Sun's Heap Analysis Tool (HAT) for Analysing Output from hprof, http://java.sun.com/developer/onlineTraining/Programming/JDCBook/hat_bin.zip
[10] JTracer Tool, http://amslib.free.fr/
[11] Java Profiler J-Sprint, http://www.j-sprint.com/
[12] JProfiler, http://www.ej-technologies.com/jprofiler/overview.html
[13] The OptimizeIt! Performance Profiler, http://www.optimizeit.com/
[14] JTest, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest
[15] JProbe, http://java.quest.com/jprobe/jprobe.shtml
[16] JView, http://www.devstream.com/
[17] Kazi I H, Jose D P, Ben-Hamida B, Hescott C J, Kwok C, Konstan J, Lilja D J and Yew P-C 2000 *IBM Systems Journal* **39** (1) 96; http://www.research.ibm.com/journal/sj/391/kazi.html
[18] Sun Microsystems: Java Platform Debug Architecture (JPDA), http://java.sun.com/j2se/1.4.2/docs/guide/jpda/
[19] Java Management Extensions, http://java.sun.com/products/JavaManagement http://java.sun.com/j2se/1.5.0/docs/guide/management/
[20] Ludwig T, Wismüller R, Sunderam V and Bode A 1997 *OMIS – On-line Monitoring Interface Specification* (Version 2.0), Shaker Verlag, Aachen, LRR-TUM Research Report Series **9**, http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/-version-2.0.ps.gz
[21] Wismüller R Trinitis J and Ludwig T 1998 *Proc. Euro-Par'98, Parallel Processing*, Southampton, UK, LNCS **1470** 173
[22] Sun Microsystems: Java Native Interface (JNI), http://java.sun.com/j2se/1.4.2/docs/guide/jni/
[23] Gamma E, Helm R, Johnson R and Vlissides J 1995 *Design Patterns*, Addison-Wesley