Towards a Monitoring Interface Specification 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. In this paper we discuss a general-purpose, portable, and extensible approach to handle comprehensive monitoring information from a Java run-time environment, based on an extension to the On-line Monitoring Interface Specification (OMIS). The main goal of the Java extension to OMIS is to create an interface that will be used to build an OMIS compliant monitoring system as well as to adapt existing tools and to build new ones for monitoring distributed Java applications. The issues covered include enhancements at the specification level. Tools based on this framework are intended to generate problem-oriented requests and handle response information in a uniform way independent of the tools' specifics and the monitoring system's implementation.

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

1 Introduction

Currently Java has become a mature programming language with quite a large amount of libraries. Its unique feature – machine independent code – makes this language an excellent basis for distributed programming. A popular approach relies on the remote procedure call paradigm. There are a few standards [5] for RPC like RMI and CORBA which have lots of implementations in Java. The run-time environment for Java programs is the Java Virtual Machine (JVM). There are a number of monitoring tools for JVM [8], especially debuggers, but they do not directly support distributed systems. These solutions are usually built on the JNI [6] and JVMDI [7] interfaces. Another class of tools, profilers which are based on JVMPI [3] interface, usually work in off-line mode and use data from traces. JaViz [4] is an example of a tool that generates profiling data from traces for RMI applications.

Our approach addresses the drawbacks of existing tools, first of all relating to poor extensibility. The tools mainly support fixed functionality and do not provide any interfaces that allow to add functions to support new features offered within a new version of the JVM. Another problem is the lack of a query mechanism that enables to define an application's entity or set of entities to be monitored. Tools only provide simple filters that are insufficient for monitoring large applications. Furthermore, profilers can only observe the applications and do not provide a facility to manipulate a program's elements.

On the other hand, on-line tools are rather few and mainly face problems when porting to Java. One of the opportunities to overcome the difficulties is an approach used in the On-line Monitoring Interface Specification (OMIS) developed at LRR-TUM [1].

In order to achieve a larger versatility, a monitoring system underlying any tool's operation should offer extensible rather than fixed functionality, e.g., in the form of a language which enables to combine services into complex service requests. In this framework, the monitoring system could play the role of an intelligent controller and mediator that gets its commands from the tool/monitor interface. Following this idea, OMIS is intended to meet the goal of building a standardized interface between application development support tools and parallel programming environments. OMIS is not restricted to a single kind of tools, especially it supports debuggers as well as performance analyzers and other tools. Extending OMIS to support Java applications enables to reuse OMIS compliant on-line tools like PATOP [9], with possible extensions for Java needs. Another benefit arising from the extended specification is a common monitoring infrastructure for building new tools that enables visualisation of running systems, dynamic reverse engineering, or monitoring of applications servers, with support for distributed operation.

For achieving such a monitoring functionality, some problems due to the differences between Java applications and the parallel systems supported by OMIS should be solved. The most important distinctions are:

- Java's approach to distributed programming is based on the RPC paradigm whereas OMIS was initially intended for message passing systems,
- we have to deal with the object-oriented nature of Java applications while OMIS supports procedural programs.

Another kind of problems relate to Java specific elements to be monitored, i.e. the virtual machine with garbage collector, thread synchronization, memory allocation, and class loading.

An infrastructure that takes into consideration the mentioned issues is presented in this paper. The paper mainly concentrates on the conceptual issues of building a tool/monitor interface rather than on implementation aspects.

The paper is organised in the following way. At the beginning we overview the principal features of OMIS. Next we discuss the main requirements for a Java extension. We then introduce the main ingredients of an OMIS extension for Java: the hierarchy of objects that can be monitored, the available monitoring services, and relationships between objects that may be used to efficiently select the objects of interest. The last part of the paper focuses on some issues of the monitoring of remote method calls.

2 OMIS

Below we briefly introduce those ideas underlying OMIS, which are important for the future discussion of our extension.

The conception of OMIS builds on the following elements: system objects, services, and relations between system objects. In accordance with OMIS, a parallel target system is viewed as a hierarchical set of objects. In its primary shape which arose from its use for the message-passing paradigm, OMIS defined five types of objects: *nodes, processes, threads, messages*, and *message queues*. Tools using the monitoring system have to explicitly attach to the nodes and processes on which they intend to operate, thereby the tools themselves are enabled to create their own scope of the system to be monitored.

For each type of system objects, a set of operations (*services*) is defined that can be used on these objects. Three categories are distinguished: *Information services* provide data on a specified object, *manipulation services* enable to change the state of an object, e.g. to stop a process. *Event services* are used to trigger a specified list of actions whenever a matching event takes place.

A central idea of OMIS is to provide an interface that allows to combine primitive services into more powerful requests. Thus, the tool/monitor-interface is based on formal language in order to achieve the needed flexibility. OMIS defines a *service request* as a string used to activate a given service or combinations of services. There are two kinds of requests: unconditional and conditional. An *unconditional requests* consist of information and/or manipulation services that are executed immediately. If a request contains an event definition, it is called *conditional* and executed whenever the defined event occurs. A mechanism called *event context parameters* allows to pass information on the particular event occurrence to the actions being executed.

At the tool/monitor-interface, all system objects are represented by elements of an abstract data type, called *token*. This abstraction provides a platform independent way of addressing objects. In order to increase the scalability of the interface, services can not only operate on single objects, but also on *object sets*. The specification of the objects a service should work on is supported by a concept called *token conversion*. Its basic idea is: rather than explicitly enumerating the objects in a set, they are specified via their relation to other objects. E.g., think of a specification like 'all processes[1] located on node 1'. Currently, conversions in OMIS are implicit and are based on the 'is located on' relation. I.e., you can simply pass a node token to a service operating on processes to denote 'all processes located on this node'. This kind of conversion is called *expansion*. Vice versa, you can also pass a process token to a node service to denote 'the node where this process is located on' (*localization*). Conversions are applied transitively, i.e. a node token passed to a thread service denotes 'all threads (in all processes) on this node'.

[1] More exactly: all *attached* processes

3 Java requirements for a monitoring interface

The main difference between the systems supported by OMIS so far and Java applications is the object nature of Java programs. Currently, OMIS mainly deals with the physical elements of the monitored system, i.e. nodes, processes, and threads. The only relationship considered between these kinds of objects is the 'is located on' relation.

In order to enable the monitoring of distributed object oriented systems, a monitoring system must include elements from the application layer as well, like *classes*, *interfaces*, *objects*, and *methods*. Access to these entities during the runtime is very desirable from the user's (programmer's) point of view. In addition, object-oriented languages lead to new relationships between system objects[2], like inheritance or polymorphism, which should be supported by a monitoring interface.

A Java-bound extension to OMIS should also include new services that allow to monitor Java and JVM specific elements like thread synchronization monitors[3], garbage collector, class loading. Moreover, Java has a nice feature: machine independent byte code that provides a possibility for dynamic class instrumentation, which should be supported by the Java extension.

4 New object hierarchy

Since our goal is to embrace all the aspects of Java applications that can be monitored, in order to meet the requirements presented in the previous section, we have extended the OMIS object hierarchy with new object types and relations. In addition, we added new services.

The new hierarchy with the supported object relations is shown in Fig. 1. In the following, we briefly describe each object type and its associated services. A discussion of object relations follows in Sect. 5.

Node. As in the original OMIS, a node represents a hardware host with a single system image. Services return information about a node and add or remove nodes from the system being monitored.

JVM. This token type represents an individual Java Virtual Machine. Since the JVM just consists of a single process, there is no need to distinguish the JVM from its only process. Services allow to manipulate the garbage collector and to obtain notifications on memory allocation events, class loading, garbage collector events, and a new thread's start.

Thread. A thread token represents a thread running on a JVM. Services provide information about threads, theirs stacks, allow to manipulate threads, and enable to observe thread synchronization.

---

[2] Note that the term *object* is used in two different contexts: On the one hand there are system objects that denote elements of the system that is monitored. On the other hand there are objects in terms of Java.

[3] Here, *monitor* means the synchronization construct introduced by C.A.R. Hoare.

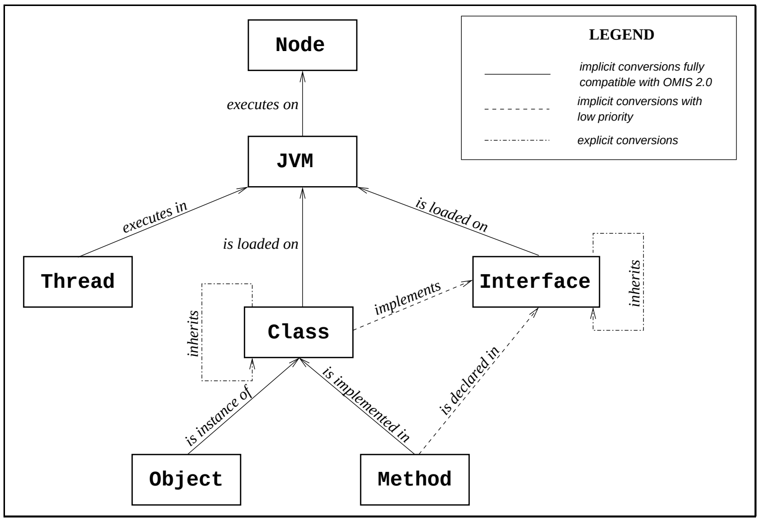

Fig. 1. The extended hierarchy of system objects for Java applications

Class. Class tokens represent Java classes loaded in a JVM. This comprises also Java abstract classes. Information services provide detailed descriptions of classes, event services can inform about class unloading and new instance creation. A manipulation service enables to exchange a class when loading, which allows to use an instrumented version of the class.

Interface. This token represents a Java interface and is defined mainly for conversion purposes. It provides services similar to those for class tokens.

Object. An object token represents a Java object. Services return information about the state of an object, can change its attribute values, or notify about events like reference creation/removal and object move/removal from memory.

Method. This token represents a simple Java method that can be invoked by threads. Services allow to obtain information about method parameters, events like method_entered/exited, or change method parameters.

5 Token conversions

An important feature introduced by OMIS is the token conversion, which provides a way for transition from an object token to other tokens related to it. As shown in Fig. 1, the Java extension introduces a couple of new relations between system objects. For clarity, the figure shows only one direction for each relation, although the inverse relations are included, too.

Introducing these new relations, however, conflicted with the implicit token conversion scheme used in OMIS (c.f. Sect. 2), since the applicable conversions are no longer unique. E.g. given the relations in Fig. 1, the conversion of a

JVM token into method tokens could result in 'all methods implemented in classes loaded on this JVM' or in 'all methods defined in interfaces loaded on this JVM'. These two sets are not necessarily the same. In order to solve this problem while still maintaining compatibility with OMIS, only a subset of the relations is unrestrictedly used for conversions (solid lines in Fig. 1). These are the relations that roughly correspond to the 'is located in/on' relation used in the original OMIS. The relations shown with dashed lines in Fig. 1 are only used for a conversion, if no unambiguity results. Thus, in the example just mentioned, the 'is declared in' relation will not be used, resulting in a unique semantics of this conversion. The inheritance relation still needs a separate handling, since without some explicit information it is not even clear whether a token should be converted at all (the source and target token type are identical). Thus, we have introduced an explicit conversion operation for this case (see below). We are currently examining how OMIS might be changed to allow a more unified handling of token conversions.

The relations introduced by the Java extension allow to request services in a very compact and efficient way, as the following examples will illustrate:

`method_get_info([`***class_id***`], 1)`
: This request results in information[4] on all methods implemented in the specified class (the class token is expanded into a set of method tokens).

`method_get_info([`***jvm_id***`]), 1`
: Returns information on all methods implemented in classes loaded on the specified JVM.

`interface_get_info([`***class_id***`]), 1`
: Results in information on the interface implemented by the specified class (the class token is converted to an interface token using the 'implements' relation).

`class_get_info([downcast(`***class_id***`,1)], 1)`
: This request returns information on the classes directly derived from the specified one. The downcast operation generates a set of class tokens which are subclasses of a given class. Its second parameter specifies the number of inheritance levels, with 0 used as a wild-card. Similarly, there is also an `upcast` operation that returns the superclasses of a class.

`object_get_info([downcast(`***i_server_id***`,0)], 1)`
: According to the Java Development Kit, all remote object must directly or indirectly implement the *java.rmi.Remote* interface. Thus, if *i_server_id* is the token representing this interface, the above request provides information on all RMI remote objects (i.e. all objects that are an instance of a class implementing an interface derived from *java.rmi.Remote*).

[4] The exact kind of information is determined by the last parameter of the `..._get_info` services.

6 Remote methods monitoring

The basic approach to distributed programming in Java is RMI. Currently, Java also supports distributed object systems like CORBA and RMI over IIOP. Each of these standards differs in details, e.g. object reference finding and offered services, but the basic mechanism of calling the remote method remains the same. This functionality is realized using *proxy objects*, which are local representatives of the server objects on the client side and vice versa.

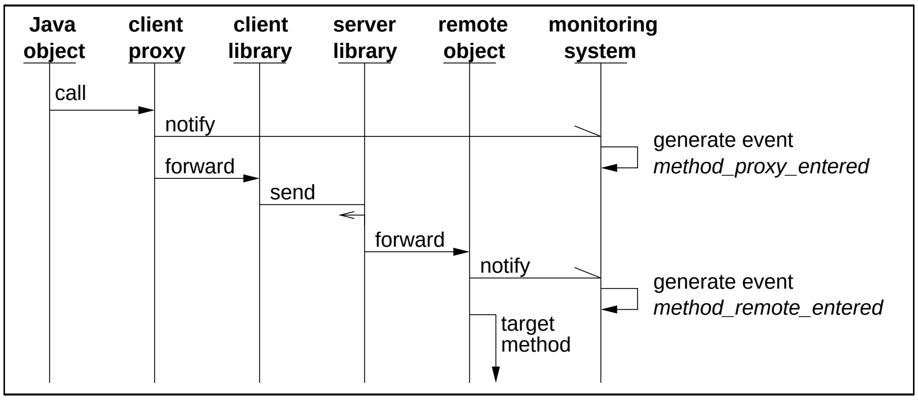

Fig. 2. Sequence diagram of the remote method invocation.

In order to provide independence from a distributed system and its implementation, the Java extension to OMIS concentrates on support for the monitoring of remote method calls. It is based on the observation of activities on the client and server virtual machine. Thereby there is a need in a mechanism that observes the operations a remote call consists of.

The Java extension to OMIS defines additional event services for method tokens: `method_proxy_entered` and `method_remote_entered`. Fig. 2 shows that the first event occurs when a remote call starts on the client side, which means that the proxy representative of the remote method has been entered. The second event occurs when the target remote method on the server side has been entered. Both services return event context parameters that provide information on a the specific event occurrence: this information includes the *method* token, the *object* token, and a remote call *tag*. The last parameter unambiguously identifies each remote call. This tag associates the method on the client side with its actual implementation on the server side. The method and object token can be used to monitor local activities during the remote call.

The described technique provides a low level mechanism to monitor remote method invocations. Future OMIS extensions can use this mechanism to provide more high level services.

7 Concluding remarks

Our work was focused on the creation of an infrastructure to support on-line monitoring of Java applications[10]. The approach we used bases on an interface specification, successfully applied to parallel environments. An implementation of the OMIS 2.0 specification (OCM) works for PVM and MPI systems. Given our experience with building and using the OCM, we strongly believe that the extension specification will turn out to be useful for Java, and for another object oriented languages. The conception of monitoring the application objects is important for object oriented systems, where objects are basic elements of a program.

Future work will concentrate on the implementation of this specification that enables to adapt existing tools and to create new ones.

Acknowledgments. The authors are very grateful to Prof. Michael Gerndt and Mr. Bartosz Baliś for valuable discussions. This work has been carried out within Polish-German collaboration and supported, in part, by KBN under grant 8 T11C 006 15.

References

1. 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`
2. Sun Microsystems: Remote Method Invocation Specification `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`
3. Sun Microsystems: Java Virtual Machine Profiler Interface (JVMPI) `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html`
4. 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`
5. Sun Microsystems: JavaBased Distributed Computing: RMI and IIOP in Java `http://java.sun.com/pr/1997/june/statement970626-01.html`
6. Sun Microsystems: The Java Native Interface Specification `http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html`
7. Sun Microsystems: Java Virtual Machine Debug Interface (JVMDI) `http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/jvmdi.html`
8. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
9. R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen: Interactive Debugging and Performance Analysis of Massively Parallel Applications. *Parallel Computing*, **22**(3), (1996), 415-442 `http://wwwbode.in.tum.de/~wismuell/pub/pc95.ps.gz`
10. 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.