Dynamic Instrumentation of Distributed Java Applications Using Bytecode Modifications

Włodzimierz Funika* and Pawel Swierszcz

Inst. Comp. Science, AGH-UST, al. Mickiewicza 30, 30-059 Krakow, Poland
funika@uci.agh.edu.pl
Tel.: (+48 12) 617 44 66; Fax: (+48 12) 633 80 54

Abstract. Java's features such as system platform independence, dynamic and network oriented architecture, robustness as well as growing number of common standards make it a language of choice for many projects. However an increasing complexity of created software and requirement for high stability and high quality of applications make it desirable for a developer to inspect, monitor, debug or in any way alter Java programs behaviour on-the-fly. The main goal of this paper is to present the design of a system for instrumenting Java classes at runtime. This system is to aid developer in modifying program by adding fragments of code at specific locations that implement some new functionality. This allows programmer to enhance classes with logging, monitoring, caching or any other capabilities that are required at run-time.

Keywords: Java, instrumentation, bytecode, J-OMIS, J-OCM.

1 Introduction

With the fast increasing power of modern computers equipped with more and more advanced processing units, memory and disk resources more complex and larger applications are created. Rapidly developing networking made distributed systems common as network-enabled computers are standard. Additionally, modern systems are often assembled from components coming from different vendors, sometimes running on quite different platforms and not fully compliant to implemented standards. This makes applications prone to errors, unpredicted behaviour, runtime flaws, deadlocks and all kinds of programmer mistakes.

Java is a leading solution for developing modern, object-oriented software, being a platform for a great number of advanced systems in various use cases both scientific projects and business oriented applications. Together with this comes greater effectiveness of software production in many domains from mobile devices to large scale systems and middle-tier integration of heterogeneous legacy solutions. Java is also successfully applied to massive computational tasks executed on a farm of high performance servers or quite contrary in distributed environment or for example grid network.

* Corresponding author.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 534–541, 2006.

The increasing complexity of developed software makes the whole process of testing, bugs detecting and fixing difficult and time-consuming. Large hardware potential made solving much complicated problems possible nowadays but this has a side-effect of making optimizing and debugging systems even more difficult. Although Java platform provides with many options of debugging, testing or in other ways producing stable applications it does not include ready-to-use kits that can be helpful when dealing with advanced, parallel and distributed programming. This leads to a demand for more sophisticated tools intended to aid the developer in creating stable end efficient software, e.g. monitoring systems, debuggers, logging systems, profilers and other. For realizing their tasks in the distributed environment, such tools might exploit various techniques and interfaces, in this paper we address one of these techniques - instrumentation.

Instrumentation does not modify original application's behaviour in terms of which instrumented program should work in the exact same way is was working. Obviously, in order to perform code instrumentation, some knowledge about how application works is indispensable. This could include some code structure details like locations of interesting fragments of code or runtime behaviour information (for example how is application using it's external resources). However these application meta-data could be collected in automatic or semi-automatic way through code (source or binary) analysis.

In this paper we address the issue of dynamic instrumentation of distributed Java applications. In order to achieve this goal, we combine techniques, which enable the analysis of application code with run-time bytecode modification and communication with distributed JVMs which are running the target application.

2 Related Work

There are other systems that exploit various instrumentation techniques in application monitoring or manipulation. One example might be the `JSpy` system for runtime analysis of Java applications [10]. It reads instrumentation specification (a set of rules in a predicate/action form) and generates logging statements in the target code. The general concept of the system is quite similar to ours, binary Java classes are also extended with additional bytecode instructions (low-level library used for that is in this case `JTrek`) and there is interoperability with external monitoring system (`JPax`). Our system is more generic while `JSpy` is entirely analysis/monitoring oriented.

Another example of instrumentation tool capable of instrumenting and modifying programs at the time of execution is the `DynInst` library [14]. It provides functions for building instrumentation tools by creating *mutator* programs that connect to the target application at runtime and change the program's behaviour on-the-fly. Although DynInstAPI is machine independent it is also highly C++ oriented and thus practically unusable for instrumenting Java applications. On the other hand, DynInst allows for fine-grained instrumentation while our system produces new versions of whole classes.

A slightly different approach is used in `Java Instrumentation Suite` (`JIS`), where instead of modifying class methods prior to executing them in JVM the

actual runtime environment is instrumented [11]. Calls to dynamic native methods are wrapped that allows to react on runtime events like starting thread or entering monitor. This makes bytecode modifications needless which is good as they are quite error prone. On the other hand, it limits the set of events the instrument can react on to those which include calling native methods from within the JVM runtime libraries. The whole system has functionality strictly limited to monitoring multithreaded applications, debugging deadlocks, etc.

An example of a tool that gives a developer a greater extent of freedom could be the `BIT` tool [12], being a framework for creating custom instrumentation applications, which provides a set of interfaces for enhancing applications with arbitrary code. BIT operates somewhere between an instrumentation system like ours and a low-level bytecode altering library. It provides no direct support for distributed applications nor monitoring - at the cost of making instrumentation process more complicated and time consuming it gives almost unlimited power to the programmer when it comes to inserting custom code at a fine-grained level.

3 Motivation and Goals

The main goal of our research is to design and implement a system that will support the developer in instrumenting Java applications. This support consists of automating some of the tasks that are the required steps of instrumentation process. In this way, the developer can concentrate on a higher abstraction level - the design of instrumentation elements, choice of instrumentation spots, combining additional functionality with the existing application classes to create functional execution units.

The designed system has to preserve maximum flexibility that means it must not limit instrument's functionality so that it can perform any task that the original program could do. This is to be achieved by using bytecode manipulation that allows for arbitrary code to be executed as an instrument is invoked. An alternative approach could be taken - that is to change the JVM behaviour rather than an application's code. Although this frees us from error-prone (and sometimes costly in execution time) bytecode modifications it requires a custom virtual machine. This is a major downside as we don't want to rely on a single JVM implementation [4]. Operating on bytecode has an advantage that it puts minimal constraints on the instrumentation process - no source code is needed and rewritten classes are undistinguishable from those produced by the original Java compiler. Therefore this is another goal of the work - to create methods for automatic bytecode modification from an instrumentation design, possibly with some help from third party libraries.

As mentioned above, distributed applications are increasingly widely used and thus some kind of tool support for them is highly required. An instrumentation system must be able to act in a distributed environment and communicate with remote virtual machines and instruments running on them. To achieve this, another tool is used - the `J-OCM` distributed monitoring system [1, 2]. It allows

to discover execution nodes, collect various data on running machines and what is more important - to interact with JVMs through low-level debugging interfaces. Therefore the main project goals include inspection of the possibilities to combine the generic instrumentation engine with the monitoring system.

The secondary goals include creating structures and algorithms for representing the instrumented application's code in an object-oriented fashion, examining libraries for the modification of binary classes in the context of instrumentation and creating a universal, generic model of instrumentation design - specifying common instrument features and ways of binding instruments to applications.

4 System Concept

The system being described comprises a few components. The main element of the architecture is an instrumentation engine with graphical user interface allowing for choosing classes to be instrumented, designing instruments to be applied and controlling the actual process of dynamic instrumentation. Other elements include low-level libraries for class manipulations (e.g. Bytecode Engineering Library), instruments definitions (class files realizing a proper interface and holding data about a specific instrumentation pattern) and last but not least the monitoring system for communicating with distributed virtual machines.

The process of instrumentation performed by the described system consists of several stages. The first one is choosing which application classes are to be modified and analysing their contents. A class model is created in the form of `SIR` tree which is a format for representing an applications' code structure [7]. Then it comes to the design of additional functional units called *instruments*. Each instrument implements some feature (e.g. writes out the current class field value) and is connected to program hooks which are places in the code where the instrument code can be injected. Having prepared the design of instrumentation, the developer can advance to the third stage and perform actual class files modifications. New fields are added to the classes, their internal structures are modified and new bytecode instruction sections are introduced to classes' methods - accordingly to previously crafted instruments and their binding to SIR elements.

After all classes that need to be changed are changed, they are distributed to (possibly remote) Java Virtual Machines running an application that is to be operated on. The original program classes are replaced in JVMs with their instrumented versions. Every object of instrumented class instantiated from that point on as well as static calls to class methods will possess and perform extended functionality. For adding network-awareness to the design, we used the J-OCM distributed Java applications monitoring system. It is used as a communication and transportation framework, provides information about hosts (nodes) and JVMs running the application as well as implements low-level class hot-swapping service using Java's new `(Java Virtual Machine Tool Interface)` (`JVMTI`) [5] . This is where the described instrumentation system differs mostly from traditional solutions. Typically, instrumentation tools focus on either enhancing

the application's code or interoperating with a target virtual machine, to gather required data or perform some operations. Using J-OCM in the described system gives the developer a better view on the environment where the instrumented application is going to be executed as well as allows to interact with it. Since one can write custom services in the J-OCM monitoring system, this functionality of interacting with the distributed environment can be further extended as new features are required.

The usability of the described system is not limited to monitoring or profiling only. Instruments can implement almost any kind of service including object state persistence (for example to relational database), caching with an external caching system or even workarounds or quick bugfixes of encountered application's problems. Practically, the developer has only to be careful about not damaging original program's functionality and not using excessive amounts of system resources which could affect application's performance or stability.

The architecture of the system is shown in Fig. 1.

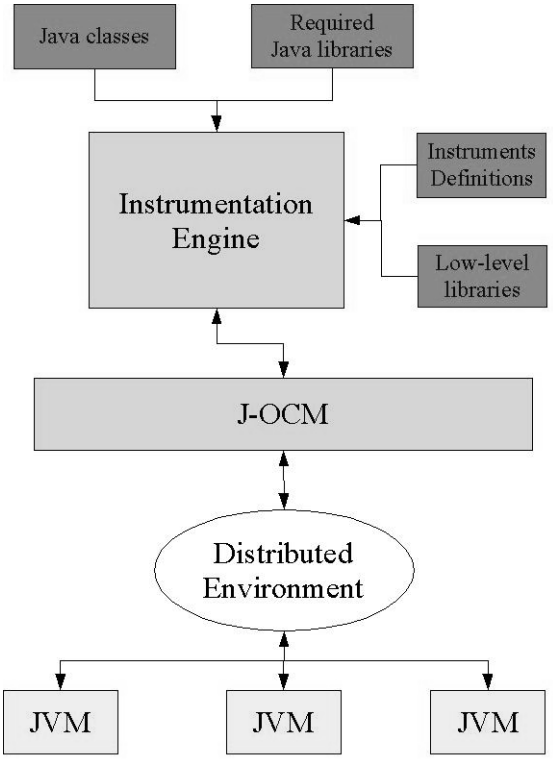

Fig. 1. Instrumentation system architecture

5 Implementation Issues

The system was implemented in Java using `Java Developers Kit` version 5.0. It was created with a free and open developer's environment Eclipse [13]. Graphic user interface (`GUI`) uses `SWT` (`Standard Widget Toolkit`) library which is an alternative to standard libraries included in Java distribution such as `AWT` (`Abstract Windowing Toolkit`) or `Swing`. Alternatively, a scripting approach would be possible, but we believe that it's easier for the user to interact with graphical interface. For example it is more convenient to examine the application's structure in a point-and-click widget than to read large XML file [6], also joining instruments with application elements is less error-prone and more user-friendly when done using a GUI. Class introspection and manipulation was performed mostly with `Java Object Instrumentation Engine` [9] v. 0.9 beta.

Like in many other applications, XML format was used for storing instrument definitions, SIR structures, and application configuration between sessions. In our project, however, instead of using the standard `SAX` or `JDOM` parser, an alternative solution is chosen. `Castor` tool was used to transform plain Java objects into XML structures based on the mapping files created by the developer [8]. Castor was chosen because it frees from writing custom XML processing code. Keeping the conversion policy in an XML file makes it also more maintainable and easier to apply any potential changes.

The most challenging task was to implement bytecode modifications to be performed based on instrument definitions and the original class structure. Not only bytecode sections have to be changed but also other static structures of the class like the constant pool containing all constant values the class uses or the stack size field that could result in a runtime error if set to an insufficient value. The actual bytecode instructions inserted have also to be valid and correct - not only from the syntax point of view but also should not collide with original instructions, change a field or variable values or in any other way alter the original runtime behaviour. A small mistake in an instruction parameter, index argument or similar details will end up in crashing VM or will cause the class verifier to refuse loading the instrumented class. Unfortunately, there is no easy way to debug instrumented classes at run-time. There is no source code for the instrumented version of the class and instrumented classes contain no valid line numbering information. What is even worse a debugger cannot be used in case of a low level class validation error or runtime VM crashes, at least no debugger capable of warning in such conditions is known to the authors. The most effective yet inconvenient way of debugging is bytecode instruction analysis. By comparing the original bytecode with the bytecode of instrumented classes, it is possible to reveal what instructions were added by the instrumentation system - it is up to the developer to find what's wrong with it.

6 Case Study

For testing purposes, a few instruments were created (designed as Java classes implementing proper interface) and later instantiated in the instrumentation

system and applied to simple test classes and methods. Later, bytecode implementations were compared between original and modified versions. The instruments used in these tests were:

- `HeapMemoryInstrument` - measures the amount of heap memory available at the time it is invoked. It can be configured to react on any runtime event but its most natural use is to inject such a measurement into those places in application code where new objects are created.
- `MethodTimeInstrument` - is intended to react on the `method execution` and `return from method` events. The first one causes instrument to get a current system time and store it as a field value. The second event makes the instrument check the system time again and compare it to a saved value. In this way, one can calculate the time spent in a method.
- `VariableWatchingInstrument` - enables to monitor changes to a variable value or class/object field value.
- `ExceptionLoggingInstrument` - reacts on the `throw exception` runtime event, helpful in debugging applications; with a few enhancements it can be used to preserve important data when a system error occurs.
- `GarbageCollectingInstrument` - is more of a utility than a monitoring probe, forcing garbage collection in VM.

The system proved to be producing a code that is working stable and correct. Instrument implementations were injected into the original code and actually executed by VM just as they were written in the original source code and compiled with the rest of the class. The user is enabled to define instrumentation with GUI and observe the whole instrumentation process with it.

7 Conclusions

While most of the above research goals have been achieved, some work still remains. The dynamic class reloading of classes after bytecode instrumentation via monitoring system is left for future development.

The instrumentation system discussed in this paper is found to be capable of performing all the tasks mentioned above with the exception of distributed class redefinition. First of all, it allows for the in-depth introspection of binary classes of the instrumented application and the creation of a model representing its structure. It provides an easy and extendable way of defining instruments as Java classes which implement a special interface. The instrument can perform arbitrary actions and what actually is done in the instrument depends only on the instrument creator. Binary modifications are performed fully automatically and the user performing instrumentation does not need to know any bytecode-level details.

As for examining the possibilities of combining the generic instrumentation system with J-OCM - using techniques defined in the J-OMIS specification [3], monitoring system services can be defined and called to perform on-line dynamic instrumentation through Java agent plugged into the target VM. This requires,

however, knowledge about how J-OCM works as well as some native coding for implementing services.

Instrumentation is often used as a means of monitoring. In this context, instead of actually modifying classes low-level interfaces provided by the `JVMTI` can be used to obtain information about the running application and even perform a sort of instrumentation for the purposes of gathering data to be utilized by tools. The general concept stays the same with the difference that the J-OCM system requests are not used for class redefinition but rather for calling proper tool interface methods via the agent.

Acknowledgements. This research was partially supported by the EU IST K-Wf Grid project and the AGH grant.

References

1. M. Bubak, W. Funika, M.Smętek, Z. Kilianski, and R. Wismüller: Architecture of Monitoring System for Distributed Java Applications. In: Proc. EuroPVMMPI'2003, LNCS 2840, pp. 447-454, Springer Verlag, 2003
2. M. Smetek: OMIS-based Monitoring System for Distributed Java Applications, M.Sc. Thesis, AGH, Krakow, 2003
3. Bubak, M., Funika, W., Wismüller, R., Mętel, P., Orłowski. Monitoring of Distributed Java Applications. In: Future Generation Computer Systems, 2003, no. 19, pp. 651-663. Elsevier Publishers, 2003
4. Tim Lindholm, Frank Yellin: The Java Virtual Machine Specification, 1999 `http://java.sun.com/docs/books/vmspec/`
5. JVM Tool Interface: `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`
6. W3Org: Extensible Markup Language (XML) 1.0 (Second Edition) `http://www.w3.org/TR/REC-xml`
7. C. Seragiotto, Jr., Hong-Linh Truong, B. Mohr, T. Fahringer, M. Gerndt: Standarized Intermediate Representation for Fortran, Java, C and C++ (to be published)
8. S. Gignoux, K. Visco: Castor XML Mapping `http://www.castor.org/xml-mapping.html`
9. G. A. Cohen, J. S. Chase, D. L. Kaminsky: Automatic Program Transformation with JOIE, 2002
10. A. Goldberg, K. Havelund: Instrumentation of Java Bytecode for Runtime Analysis, 2003 `http://ase.arc.nasa.gov/havelund/Publications/jspy-final.pdf`
11. J. Guitart, J. Torres, E. Ayguade, J. Oliver, J. Labarta: Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications, 2000 `http://citeseer.ist.psu.edu/guitart00last.html`
12. H.B. Lee, B. G. Zorn: BIT: A Tool for Instrumenting Java Bytecodes, 1997 `http://citeseer.ist.psu.edu/lee97bit.html`
13. Object Technology International, Inc.: Eclipse Platform Technical Overview, 2003 `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`
14. B. Buck,J.K. Hollingsworth: An API for Runtime Code Patching, 2000 `http://www.dyninst.org/`