Java Bytecode Instrumentation: An Introduction

This post is not a usual one since I simply want to address a technical question: "what is Java bytecode instrumentation (BCI)" and also explain what can and can't be done with BCI regarding the problem of transaction tracing. It's just that I've been asked about it again and again, and there is a real confusion out there in the market. Vendors that ONLY do BCI (e.g., CA Willy, dynaTrace, AppDynamics, etc.) are claiming to be a transaction management solution, although there are limitations to what they can do in Java environments, and they have zero visibility to non-Java topologies.

BCI is a technique for adding bytecode to a Java class during "run time." It's not really during run time, but more during "load" time of the Java class. I'll explain: Java, for those who are not familiar, is a fourth generation language, which means you write Java code—e.g., create a *.Java file—you compile the code—e.g., creating a *.class file, which is written in bytecode, and when you execute it, an interpreter—the Java.EXE—is responsible for actually executing the

commands written in the bytecode format within the *.class file. As with any interpreter, since we are not dealing with real object code, one can manipulate the actual code written in the executed file.

For example, let's say you want to add functionality to a Perl/PHP/JSP/ASP code—that's easy. You could simply open the file in a text editor, change the code, and next time it was executed it would behave differently. You could easily write a program that changes the code back and forth as you wish as a result of some user interface activity.

With bytecode it's the same concept, only a bit trickier. Try to open bytecode in a text editor—not something you want to work with...but still possible ☺. Anyhow, the way to manipulate the actual bytecode is by intervening during the class loading flow and changing code on the fly. Every JVM (Java Virtual Machine) will first load all the class files (sometime it will do it only when really required, but that doesn't change the following description) to its memory space, parsing the bytecode, and making it available for execution. The main() function, as it calls different classes, is actually accessing code which was prepared by the JVM's class loaders. There is a class loader hierarchy, and there is the issue of the classpath but all that is out of the scope of this post...So the basic concept of bytecode instrumentation is to add lines of bytecode before and after specific method calls within a class, and this can be done by intervening with the class loader. Back in the good old days, with JDK <1.5, you needed to really mess with the class loader code to do that. From JDK 1.5 and above, Java introduced the Java agent interface, which allows writing Java code that will be executed by the class loader itself, thus allowing the manipulation of the bytecode within every specific class, and making the whole process pretty straightforward to implement, thus the zillion different products for Java profiling and "transaction management" for Java applications.

Next up: What does bytecode instrumentation have to do with transaction tracing?

Posted on November 16, 2015

← Application Topology Mapping

Apple iOS Encryption →

Transaction Management

Transaction Tracing

Archive

May 2018

April 2018

March 2018

February 2018

January 2018

December 2017

November 2017

October 2017

September 2017

July 2017

April 2017

March 2017

January 2017

December 2016

Learn

- About Correlsense
- Meet the Team
- News
- Testimonials
- Our Partners
- Customers
- Board of Directors

Engage

- Product – SharePath
- Product Features
- Case Studies
- Resources
- Blog
- Get SharePath Free

Follow

Search …

GO

Privacy Policy