A Java Instrumentation-based Analysis Approach for the Dynamic Behaviors of J2EE Applications

Yuehong Liao, Jiang Guo
Department of Computer Science
California State University Los Angeles
Los Angeles, USA

Xichun Pei
Computer Science and Information Engineering
Inner Mongolia Agriculture University
Inner Mongolia, P.R. China

Abstract – In the last 5 years, J2EE has been widely used in the software system development, so the performance and related dynamic behavior of J2EE applications are more and more important to the software development. However, it is difficult for the software developers to capture, measure, and optimize the performance of J2EE applications without the supports of the software tools. This paper presents a novel approach to analyze the performance of the J2EE applications based on the Java instrumentation. We also discuss a software tool – JPManager – a J2EE performance management system based on our approach.

Keywords: J2EE, Performance Management, Java Instrumentation

1 Introduction

In the last decade, Java has been becoming the language of choice for developing and deploying enterprise scale server-side applications. Java 2 Platform, Enterprise Edition (J2EE) technology is becoming the standard of the development environment for server-side Java applications. The J2EE platform provides solid baseline standards on various functional components or containers for presentation and business logic with communication links to client-side presentation, as well as back-end database and legacy systems. It also offers communication links to other remote J2EE systems [1].

J2EE architectures share a basic N-tiered framework. With clients on a front tier, data sources on a back tier, and one or more middle tiers – or middleware – in which to develop enterprise applications, today's multi-tiered infrastructures are built to support distributed applications. Applications are deemed "distributed" when they run on technologies across the IT landscape, such as back-end mainframes, mid-tier application servers, and front-end Web servers and clients.

In general, business logic is a set of guidelines for meeting the discrete functional needs of a business domain – such as retail or finance. To establish business logic for a business function, developers using J2EE component-based technologies adopt an object-oriented approach. Such an approach entails decomposing a function into a set of components. Because business logic dictates the structures and behaviors of business objects as well as the conditions that must be met for business functions to run to completion, its vitality is indelibly linked to the vitality of the business. When business logic fails to perform, business functions halt – client accounts stop updating, orders stop entering – and the business as an entity fails to perform.

Similarly critical to the business's performance is the performance of the application server, the site of business logic deployment. Physically and functionally central to the multi-tiered IT infrastructure, the application server is the distributed application's conduit between the infrastructure's presentation and database layers, and the client device's shield from the enterprise's inherent complexity. In the Web space, where high traffic intensity and concurrent user requests are common and the application server supports distributed applications by rapidly and accurately processing user requests into business logic and facilitating the concomitant back-end database requests.

So, the application server plays a crucial role in business processes: it serves as the focal point of a business's lifeblood: real-time transactions. When the application server slows because of poor performance, the business it fuels slows as well; business logic processes fail to run to fruition; the business itself fails to transact. The application server, therefore, like the business logic it houses and the distributed application it supports, is a key indicator of business performance.

J2EE technology and the associated application servers allow Java applications to be deployed in a wide range of configurations with varying performance and scalability needs. As the increase of the adoption of J2EE technology for enterprise applications increases, the concerns of the performance, scalability, reliability, and availability of J2EE systems also increase. Due to the size and complexity of most enterprise applications, performance management strategies...
must include the use of tools and automation for locating and analyzing J2EE performance bottlenecks.

J2EE is critical to, and corresponding prevalence in, Web enterprises lends urgency to the task of attaining J2EE performance visibility and granularity. Without this perspective and this level of detail, businesses cannot obtain the information with which to foster company-wide, collaborative problem solving. They also lack a vehicle to facilitate troubleshooting communication between software developers and quality assurance staff. More insidiously, businesses uninformed about J2EE performance do not have a cost-effective means for identifying the issues truly impeding their Web enterprise performance.

Therefore, monitoring, analyzing and managing the performance of J2EE components and application programming interfaces (APIs) has, to date, been an important task. Understanding which components and APIs are important to manage requires in-depth knowledge. Add to that the fact that hundreds and even thousands of components and APIs must be tracked continually – all of which execute in seconds or even milliseconds – and the task of manually monitoring J2EE environments becomes unfeasible for the administrator or infrastructure manager [2]. Utilities have been developed to help ease the pain, but to date they have been largely focused on preproduction development processes such as code profiling. Our approach provides a solution of the problem by analyzing the performance and related dynamic behavior of J2EE applications based on Java instrumentation.

2 Performance Analysis of J2EE Applications

The aim of J2EE performance analysis is to evaluate performances of the J2EE application and all the back-end systems (DB and application server) and change the load. Performance analysis activity improves system performances according to a defined load and identify where response time is too high.

Using a complete approach means evaluating performances provided by every element of the chain from the user to the back-end systems. End-to-end monitoring means to measure the performance and response time of the J2EE application from front end to back end. They provide reports about the end-to-end response time (i.e. the response time experienced by the user). There are two different kinds of end-to-end monitoring tools: intrusive (Java Instrumentation) and non-intrusive (Ghost Instrumentation) [3].

Ghost Transaction – To manage the system performance, the users must know what is going wrong and how to get it fixed. This begins with a detection mechanism. The only way to know the end-to-end performance of an application is to measure the end experience of the user, which means placing detectors on the n-tiers of the J2EE systems.

To measure applications from the outside in, a major technique has been to use ghost transactions executed from a set of Java-based agents distributed across the J2EE systems. These ghost transactions actually exercise the Web application from afar, automating browser interactions and measuring the time that it takes, from the client’s perspective, to execute each transaction, move from URL to URL, or to go through the aggregation of click-through transactions that comprise a single business process.

Ghost Transaction is based on real clients installed on the Internet according to the real geographical distribution of users (this information can be extracted by a Web Log Analysis tool). These clients are trained to execute the Web performance measurement: a complete approach same transaction and activity of users, so they act as a probe on the Internet. This approach is completely “non-intrusive”, but needs a distribution of clients that should reflect the user distribution.

Java Instrumentation – Another way to secure measurements is to instrument the application itself, placing Java agents at key process points to yield fine grained measurements of application performance down to the sub-process levels. The advantage of this approach is that one can make fine-grained measurements very deeply into the application structure, such as down to the method level of an application, or even down to the line level. With the ghost transaction method, one can measure the time between the appearance of Web pages or the transition between URLs, not the creation times of the subordinate components of the Web page creation [4]. The disadvantage of the Java instrumentation is that the application itself must be instrumented. There is the potential for the instrumentation of the application to add to overhead in the application. The counterbalancing upside, besides granularity of bottleneck detection, is that no dummy test cases are required since the agents are riding on top of the actual functioning of the application, taking their measurements from real transactions and user can measure the real response time of J2EE application.

Many people have done a lot of efforts on building the J2EE performance management tools to support the software system performance optimization [5]. However, few reports in literature can be found about
using Java instrumentation approach to analyze the performance and dynamic behavior of J2EE applications.

3 The Approaches of Java Instrumentation

J2EE performance analysis requires the gathering of different kinds of information on the runtime behavior of a program when run against a set of inputs. Examples of such information are coverage data for various code entities, such as method, traces at different levels of abstraction, and program profiles.

Depending on the class it chooses to instrument, we can adopt one of two instrumentation methods: (1) Class Intercepting or (2) Method Hijacking. In general, we can use agents to instrument J2EE applications by changing the hierarchy of class inheritance or by adding new methods and fields to classes.

Class Intercepting – Every class has a “parent” or governing class, which means the classes in Java form a class hierarchy. In the Class Interceptors scenario, we can change a target class’s parent class but retain the intent of the target application. This method entails changing the name of the parent class in the file structure that represents the target class’s parent class in JVM memory. The new parent class is a subclass of the original parent class. When the subclass calls into inherited methods, the instrumented code of the new parent class activates.

Method Hijacking – Method Hijacking allows us to instrument a class’s methods and fields directly, regardless of class hierarchy. To make this method effect, when a class containing an instrumented method loads, the agents copies the instrumented method into the target class and renames the new method so that it, and not the original method, invokes. We can employ this method for such J2EE components and APIs as JSPs, EJBs because these components and APIs use Java interfaces to define the standard calling convention.

Java Instrumentation – Java Instrumentation is the addition of byte-codes to methods for the purpose of gathering data to be utilized by tools. Since the changes are purely extra additive, these tools do not modify application state or behavior. Examples of such benign tools include monitoring agents, coverage analyzers, and event loggers [6]. Each type of instrumentation corresponds to a specific code construct [7]. The set of code constructs that we instrument includes method calls according to the user specified in the instrumentation profile. We refer to the code constructs that we can instrument as instrumentable entities, such as method entry, method exit, before the method call, after the method call etc.

In general, when collecting dynamic information about a set of executions, we are interested in collecting information for some specific entities in the code (e.g., method calls and paths) and in a subset of the program (e.g., in a specific module or set of modules) [8]. Our approach lets the user specify, using instrumentation tasks, (1) the types of entities to instrument, (2) the parts of the code in which those entities must be instrumented, and (3) the kind of information to collect from the different entity types. Above information is specified in the instrumentation profile as a XML profile.

An instrumentation task for a given system is divided into two main parts. The first part specifies the part of the system being instrumented, in terms of classes or methods. The second part specifies which types of entities must be instrumented, what kind of information should be collected from each entity type, and how such information should be processed. An instrumentation task can also, in turn, consist of a set of instrumentation tasks. Composite instrumentation tasks give users the possibility to collect different kinds of information from different parts of the system. Composite instrumentation tasks also allow for processing differently the information collected for different parts of a system [9][10].

4 Four Steps of Java Instrumentation

We implement the Java instrumentation at the byte-code level by using the Byte-Code Engineering Library (BCEL) [11]. In our approach, the instrumentation tasks are specified using XML files. We use a custom class-loader that invokes the agents on classes that must be instrumented, according to the specified instrumentation task. Here, classes that must not be instrumented are loaded normally. As the JVM executes the instrumented parts of the program, dynamic information is collected in the system. Then, we can process and store the dynamic information.

For each instrumentable entity, the agents can provide a variety of information associated with that entity. For example, in the case of a method call, we can report the target object and all of the parameters passed to the called method.

We take 4 steps to implement data collection of dynamic Java byte code instrumentation (see Figure 1).
Run Data Collection Agents in a Standard JVM

Patch the ClassLoader based on the Instrumentation profile

Create a New JVM with Patched ClassLoader and Run J2EE Application in the Patched JVM

Extract Data and Send them to JPManger Server

Figure 1. The Working Steps of the Java Instrumentation

(1) Run our Data Collection Agents (DCAs) in a standard JVM. At this step, a standard JVM is a started and the DCAs are running in the JVM.

(2) Patch the ClassLoader based on the profile defined in a XML file by the user. At this step, we need take two actions.

First, we can use `ClassLoader.getSystemClassLoader()` to find the right ClassLoader to patch. A class loader is an object that is responsible for loading classes. The ClassLoader class uses a delegation model to search for classes and resources. It is a class defined at: `java.lang.ClassLoader`. Each instance of ClassLoader has an associated parent class loader. When requested to find a class or resource, a ClassLoader instance will delegate the search for the class or resource to its parent class loader before attempting to find the class or resource itself. The virtual machine's built-in class loader can serve as the parent of a ClassLoader instance. Here, we use `getSystemClassLoader()` method to get the system class loader for delegation.

Second, we need to customize the ClassLoader based on the Java instrumentation profile defined in a XML file by the user. We use `ClassLoaderPatcher()` to modify the JVM's built-in ClassLoader so that ClassLoader will work in the way that we want.

(3) Create a new JVM with the patched ClassLoader and run the J2EE application in the patched JVM. At this step, we use `vm.redefineClasses()` to create a new JVM. In fact, it puts the patched ClassLoader into the running JVM.

Interface VirtualMachine is defined at: `com.sun.jdi`. The JDI provides explicit control over a virtual machine's execution. We Interface VirtualMachine to access to global VM properties and control of VM execution. Instances of this interface are created by instances of Connector. Connector is a method of connection between a debugger and a target VM. A connector encapsulates exactly one Transport that is used to establish the connection. VirtualMachine's redefineClasses enable us to make a "HotSwap" class file replacement. HotSwap adds functionality to allow a class to be updated while under the control of our data collection agents. Here, we use redefineClasses to replaces the class ClassLoader's definitions. This means the standard ClassLoader is substituted by the modified ClassLoader (we call it patched ClassLoader) in the running application. This function does not cause any initialization except that which would occur under the customary JVM semantics. In other words, redefining a class does not cause its initializers to be run. So, the execution of the ClassLoader will not be interrupted. After this step, our patched ClassLoader will load all the classes. This means all the behaviors of the classes are under our control. It is ready for us to let the instrumenter to implement the instrumentation of the Java byte code.

After standard ClassLoader is replaced by the patched ClassLoader, the target VM machine has to be re-launched. First, we use the method `virtualMachineManager()` of Bootstrap to get the virtual machine manager. Bootstrap is a class defined at `com.sun.jdi.Bootstrap` is an initial class that provides access to the default implementation of JDI interfaces. Second, we use this class to access the single instance of the VirtualMachineManager interface. VirtualMachine Manager is manager of connections to target virtual machines. After we get the VirtualMachine Manager, then we can use its method `launchingConnectors()` to get the connectors to launch a new target VM and immediately create a VirtualMachine mirror for it. Third, we use the launch method of the interface LaunchingConnector to launch the application and connect to its VM.

(4) A Patched JVM extracts and sends the application execution information to the JPManger server. This step is straightforward. While the JVM ex-
JPManger – A Java Instrumentation-based Performance Analysis Tool

JPManger is a performance management system specifically designed and developed for J2EE applications by Yuehong Liao and Dr. Jiang Guo at California State University Los Angeles. JPManger provides the critical functions required to manage performance on the J2EE-based applications. JPManger employs a multi-tier, fully distributable architecture to provide the scalability to meet the variable needs of Java applications and deployments. It lets users collect runtime information with limited effort because it allows for easily specifying (1) which types of entities should be monitored at runtime, (2) in which parts of the code such entities should be monitored, (3) what kind of information should be collected for each entity, and (4) how to process the information collected.

Figure 2 is an execution of JPManger. An overview of the application execution can be seen in this part of the JPManger. The JPManger uses two windows to display the dynamic information of the application execution. The left pane is the action pane, which is used to display the methods of the application that have been executed. Each method can be called more that once. So, when the user clicks the plus (+) icon in front of the method, the number will appear. Its value is from 0. 0 means that the method is first time called. When user clicks the method calling number, the related information will show up in the right pane. It is a sequence diagram of the called method.

The sequence diagram in the right pane shows the flow of the logic with the application systems in a visual manner [14], enabling the user to find which method takes too long time. In this pane, the sequential nature of the logic is shown via the ordering of the messages (the horizontal arrows). The first message starts in the top left corner, the next message appears just below that one, and so on. Time is depicted vertically, the columns are methods of execution. “>” with bold black label marks the start of the execution of the method and “<” with light green label means that method has finished and return the control. The method executions are colored in green and red. Green means the execution of the method is within the threshold and the red means it takes too long time and is the performance bottleneck of the application. Its performance should be improved.

The top line of the sequence diagram pane lists the classes that send and accept the messages. For example, in Figure 2, class org.apache.struts.ActionServlet sent a message getServlet-Context to class javax.servlet.GenericServlet. The execution of method getServlet-Context() is green and its performance is acceptable. However, in Figure 3, the method loadUsers of SetupServlet took 3579 ms to finish execution (part showed in the figure). Its execution is red and its performance should be improved. As we can see in Figure 3, during the execution of loadUsers, class SetupServlet sent a message getConnection to class beans.jdbc.ConnectionPool and the later sent a message isConnectionValid to class beans.jdbc.DbConnectionPool.

Since the instrumentation process requires adding new method calls to a class file, class and method names as well as other constants about these methods need to be inserted to the constant pool table. The constant pool table is a place where different string constants, class names, field names, and other constants are stored for each class file. To support the ability to add a method call before or after a certain entity (e.g., method, basic block, etc.), the descriptor, the class name, and the method name of the method being inserted need to be present in the constant pool table of the code being instrumented. The constant pool table is also used as a place to store arguments to the analysis methods.

In the JVM, there are several method invocation instructions such as invokevirtual, invokevirtual, invokespecial, and invokeinterface. In DCA, analysis method calls are inserted by using the invokevirtual bytecode instruction and therefore, analysis methods have to be static. This decision implies that objects cannot be associated with these methods. The invokevirtual bytecode could have been used, but to keep things simple, only the invokevirtual instruction is used. To use invokevirtual, more complex sequences of bytecodes would have to be inserted in the instrumented
program because an instance of the class would need to be created and manipulated.

6 Conclusions

Intelligent J2EE instrumentation—the processing of using source code as a vehicle for tracking J2EE functioning—coupled with enterprise-wide correlation of metrics from all IT components, is crucial to maximizing the performance of J2EE application server environments. Maximal performance in J2EE application server environments is, in turn, irreplaceable in the equation yielding top business performance [15].

![Figure 2. Execution of the WEB Store application and without Bottleneck](image1)

![Figure 3. Execution of the WEB Store application and with Bottlenecks](image2)

Performance management solutions for J2EE applications must meet many demanding requirements.

Users require a full range of performance management functions addressing the specific challenges posed by
Rapid application development and deployment means that any solution must include extensive automation capabilities, and the product must be easy to install and configure, with minimal IT involvement [17]. Additionally, instrumentation technology is required to function completely with third-party components as well as custom developed software.

JPManager is an instrumentation-based performance management tool for J2EE applications. It supports both monitoring and analysis. It presents current application performance state in such a way that problems can be rapidly detected based on the sequence diagrams. In fact, JPManager not only can be used in the performance management of J2EE application, but also can be used in any Java-based systems, such as pure Java-based database system – IBM’s Cloudscape. JPManager is a system still under development. It will be extended to include state monitoring and thread monitoring so that it can be used in embedded Java systems.

References


