ProbeMeister

Distributed Runtime Software Instrumentation

Paul Pazandak and David Wells*
{pazandak , wells@objs.com}
Object Services & Consulting, Inc. Baltimore, MD

Abstract

Dynamically deployable software probes facilitate ad hoc runtime application monitoring and troubleshooting. Using the latest features of Sun Microsystems' JDK 1.4, we have built a prototype system called ProbeMeister that can attach to multiple remotely running applications, and effortlessly insert software probes to gather information about their execution. This information can be used to effect changes within the running applications to recover from unanticipated failures, or to improve their operation. While ProbeMeister is useful during software development and testing, its advantages are better realized after the software is up and running at the users' sites.

1 Introduction

Software probes enable the monitoring of running applications. Current probe tools are primarily designed for software testing: developers insert probes into their code or underlying OS during testing to emit data to help locate bottlenecks, memory leaks, bugs, or to visualize code coverage, etc. Probes of this kind may also be left in an end-user version of an application for bug reporting: if users encounter problems at a later date, the log files generated by the probes can be sent back to the software vendor for analysis. Both types of uses require skilled programmers to place and compile probes into the application and to determine the corrective actions to be taken once the probe data has been gathered and analyzed. Required changes are made and the software is recompiled again.

Our goal in developing our own instrumentation tool was to produce a technology suitable for distributed reconfigurable component-based software - potentially widely-distributed applications whose components are loaded on demand. Such systems are difficult to extensively test prior to deployment, partially because their environment is often immense (think Internet scale) and constantly changing. Moreover, the components may be developed by separate companies and typically evolve independently of each other, increasing the probability that problems will arise.

To probe these kinds of systems, our tool would need to be able to connect to running applications and deploy probes to each of the distributed components, then gather up all probe output for runtime tool-based analyses. In conjunction with other tools, required changes would be made without recompiling or restarting the application[1].

Using the latest features of Java JDK 1.4, we have built ProbeMeister, our second-generation instrumentation tool capable of deploying probes into remotely running Java software. ProbeMeister instruments Java bytecode, and works without needing to copy supporting code libraries to the remote machines. Probes can be deployed and removed at will into any running Java application, remote or local.

ProbeMeister is being developed as part of the Software Surveyor project [1] within the larger DARPA DASADA program [2], the goal of which is to develop technology to model, monitor, and manage dynamically composed and evolving systems.

2 Overview of ProbeMeister

ProbeMeister facilitates the instrumentation of a distributed running application with software probes[2]. It accomplishes this in part by manipulating the in-memory representation of the running application. A key capability of ProbeMeister is that its extensible set of software probes can be inserted or removed at any point while the application is running. A second key capability is that it supports the instrumentation of multiple remotely running applications. Both are necessary for monitoring evolving, distributed applications; since the application's connectivity and components may change during execution, it is essential to be able to insert and manage probes in multiple remote components simultaneously.

Prior to developing ProbeMeister, we developed the Java Bytecode Instrumentor (JBCI). JBCI is a static bytecode instumentor. It requires a multi-step process of loading a class offline into JBCI, deploying and customizing the selected probe(s), saving the modified class, and then restarting the application. Removing a probe requires similar steps. What we found was that once we knew exactly where we wanted to deploy all of the probes, the process was relatively quick. However, we also found that probe placement is an intensely iterative process unless perhaps the user is also the developer. For the overall project that we are involved in it is understood that a ProbeMeister user is not always the developer, but perhaps only a skilled application user having solid but general knowledge about how the application works [3]. When the application is not performing as expected, either as determined by the user or by pre-deployed (possibly even statically deployed) monitoring probes, task-specific probes could be deployed to gather more information to determine if a given component is not working as expected. A corrective response could be to modify specific parameters in the code to tweak its behavior, or to replace the component with a more reliable or more available one.

As software developers, we felt that the probe deployment cycle was a hindrance to ad hoc exploratory probing. This was the prime motivation for replacing JBCI with the much more dynamic ProbeMeister. In moving from JBCI to ProbeMeister, we immediately enjoyed the benefits of dynamic distributed probe deployment. Not only did it practically eliminate the deployment cycle, but also the results generated by the probes could be seen immediately without having to restart the application.

Supporting easy probe placement by non-developers requires additional tools and interfaces having knowledge of the application's architectural model, that can suggest probe deployment locations (or automatically deploy probes) based upon the problems the user (or the analysis tool) wants to troubleshoot. Model-based probe deployment is the focus of another

* This research is sponsored by Defense Advanced Research Projects Agency and administered by the US Air Force Research Laboratory under contract F30602-00-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government

[1] Imagine needing to modify, recompile, and restart a vital army tank subsystem during battle.

[2] ProbeMeister also supports the dynamic creation of new classes and complete redefinition of existing classes.

aspect of the Software Surveyor project, and will be the topic of a future paper.

Finally, ProbeMeister has not been designed to compete with coverage-oriented optimization tools. These tools are certainly more efficient at this since they can statically instrument an entire application with perhaps hundreds or thousands of probes to collect performance data. ProbeMeister is geared toward targeted placement of probes to inspect (and possibly effect changes upon) a running distributed application. However, the new interfaces in JDK 1.4 now makes it unnecessary to deploy any probes into a remote application to provide coverage feedback. We just haven't focused on exposing this capability in ProbeMeister yet.

2.1 Related Work

Prior to developing JBCI (about two years ago) we performed a reasonable search and tool review of several Java (and C++) bytecode related packages (see Related Works in section 8 for the links to the mentioned Java tools). We were looking for an extensible tool that would allow us to write our own bytecode probes and deploy them. While we could have looked at source code probe deployment tools, we didn't want to limit probe deployment to applications in which we had source code access, nor did we want the overhead associated with adding a recompiling step to the deployment cycle. We found that some of the available instrumentor tools appeared to be closed, and didn't allow one to write their own probes. These tools were geared toward software profiling (e.g. JProbe, NuMega, OptimizeIt!). Some other tools were close to what we were looking for but were not extensible, no longer supported, or too costly (e.g. JOIE, Jtrek, JFParse). Yet another group approached instrumentation by providing modified or pre-instrumented JVMs (e.g. Jinsight, eTective, BCA). Finally, the last tools (e.g. BIT, Jikes, and BCEL) were simply bytecode editors. For our needs, we thought it would be more efficient to prototype our own tool using one of these editors. Jikes is the editor we integrated into JBCI.

Even though JBCI worked reasonably well as a standalone tool, it could not be used (in the next stage of the DASADA program) by other tools at runtime to deploy probes since it only supported static instrumentation. This was an obvious and significant drawback to using JBCI. Thanks to several people at Sun Microsystems, we were fortunate to get access to an early version of JDK 1.4 in which the Java debug interface (JDI) had been extended to support remote runtime bytecode modification [4]. Using JDK 1.4, we began a complete re-implementation of our tool (about one year ago), now called ProbeMeister.

We should mention that we did find some similar tools once we began implementing ProbeMeister. For example, we found an interesting product called RootCause, which offered a sort of "one-time" dynamic probe deployment by instrumenting a class as it is loaded into the JVM. Other similar products like this exist in the C++ world, such as NTWrappers, that can modify a DLL just prior to it being loaded by the OS. Of course, none of these offered the kind of flexibility we desired.

3 ProbeMeister Architecture

In this section we present a high level view of the ProbeMeister architecture. In the lowest layer, ProbeMeister has a Virtual Machine (VM) Manager that accepts or initiates connections with other JVMs via the JDK's JDI interface. Connection behavior is enabled and configured by the targeted application through command-line arguments to the Java interpreter -- the application may initiate the remote connection (as a client) to ProbeMeister (running in a separate JVM), or it may begin its execution and allow ProbeMeister to initiate the connection (acting as a server) at some later time. In either case the targeted application requires no additional code as the underlying JDI extensions manage the connection to ProbeMeister. Using the JDI interface, an application like ProbeMeister can set breakpoints, subscribe to events (e.g., class loading, method entry and exit, etc), modify methods, and even create new classes ones on the fly.

If the application initiates the connection to ProbeMeister, the connection is established before the core JDK classes have been loaded, and therefore also before its main() method is invoked. This enables probes to be deployed before any of the application code has been invoked. It therefore allows the probes to capture the application's entire behavior from the beginning. When the connection to the application is opened, ProbeMeister stops the remote JDK's execution prior to loading of any of the application's classes. This enables the user to instrument any of the core JDK classes (such as java.io.File to monitor file access) and therefore capture all application activity. The user could also schedule probes to be automatically deployed as the application's classes are loaded.

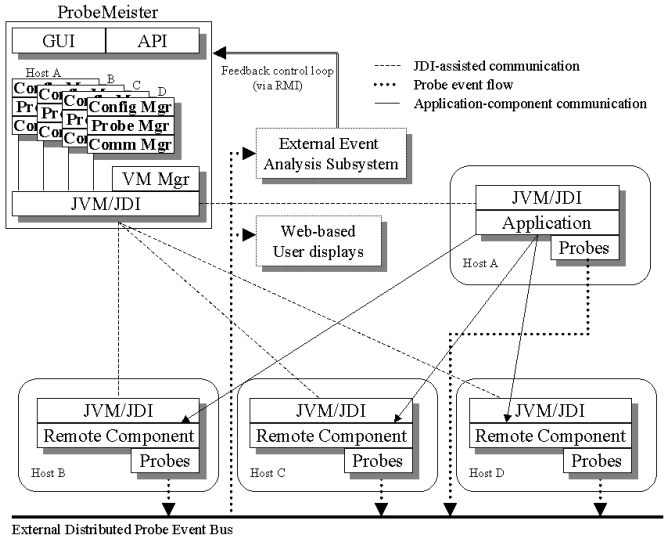


Figure 1. ProbeMeister Deployment Scenario

While having each application connect to ProbeMeister is convenient, we feel that it is an unreasonable constraint. ProbeMeister should be able to be activated on-demand, as needed. JDI-specific Java interpreter command line arguments allows an application to accept a remote connection at any point during its execution. Using this approach, a ProbeMeister user (or external tool) may request a connection to an application by specifying its address and port (as defined in that application's command line arguments). After a connection is established ProbeMeister may be used to deploy probes.

Once an application is connected to ProbeMeister it is assigned a set of components: a Connection Manager that manages the communication with the remote application; a Probe Manager that controls the creation and deployment of probes; and, a Configuration Manager that provides control to deploy or remove several probes (a probe configuration) simultaneously.

Finally, in the highest layers, ProbeMeister provides a user interface as well as programmatic interfaces so other tools can control it (locally or remotely). The following describes each of the connection-specific components in more detail. Figure 1 shows the architecture of ProbeMeister in a typical deployment scenario.

In the depicted scenario, ProbeMeister is connected to four remotely running Java Virtual Machines (JVMs) that make up a given distributed application. Probes that have been inserted, when invoked, emit descriptive events to the Siena Distributed Event Server [5] event bus[3] for remote delivery to interested consumers. The emitted events are consumed by a separate external system (tools under development) that analyzes the events, generates user consumable output (status or warnings), and may also feedback into ProbeMeister by dictating further probe reconfiguration. Of course, ProbeMeister can be used for manual user-driven monitoring and analysis. For this, we have built web browser-based HTML and XML user displays that collect and categorize probe events, and display event details[4]

3.1 Communication Manager

When a connection is established between the VM Manager and a remote application the connection is assigned to a communication manager. The Communication Manager manages the connection with a distributed application or component via the JDK 1.4 JDI interface. It provides the routines for accessing the remote classes, modifying the classes, and monitoring the state of the JVM. All calls affecting the remote JVM pass through this component.

3.2 Probe Manager

The probe manager controls the insertion and removal of application probes. Probes may be inserted when a class is first loaded, before any of its methods are invoked, or at any point after that. Insertion involves loading the chosen class' Java bytecode from its class file, modifying the selected method by inserting the probe-specific bytecode, and writing out the modification to the remote JVM using the JDI API call:

```
VirtualMachine.redefineClasses().
```

ProbeMeister currently uses the Bytecode Engineering Library[6] to modify Java bytecode[5]. To understand the minimum cost to deploy a probe, it takes on the order of 20 milliseconds to create a basic probe, modify the bytecode, and invoke `redefineClasses()` on a small locally running application. It took an average of about 250 milliseconds to deploy the same probe on the same application running in Baltimore with ProbeMeister running in Minneapolis[6]. This has been more than adequate to date given that we have created on the order of no more than tens of probes per remote application.

`redefineClasses()` takes as an argument the entire modified bytecode of the class. Once invoked, it replaces the class definition in the remote JVM. However, only new invocations will execute the new version of a modified method; currently running invocations will run to completion using the older code. And while the JDI specification allows for considerable changes to the bytecode (e.g. new methods, new attributes, completely redefined class, etc), it is up to the individual JVM implementations. At this point, Sun's JVM implementation only supports method modification. Once ProbeMeister modifies a class, since the modifications are transient a copy of the modified class bytecode is retained and used as the basis for any further probe insertions or deletions. A detailed description of the supported probe types is presented in a later section.

While ProbeMeister's Probe Manager has been designed to support the management of heterogeneous (multi-language) probes, thus far we have focused exclusively on supporting dynamic (or runtime) Java bytecode probes. Runtime probes are inserted while the application is running, while static probes are inserted when the application is offline. Runtime probes are transient by default, and are lost once the application terminates; static probes are persistent by definition. To make runtime probes persistent, the in-memory modifications need to be saved back to disk in Java classfile format. The modified classfiles can replace the original classfiles, or be stored separately (however, the configuration manager eliminates the need to do this, as described in the next section).

Finally, while ProbeMeister maintains a list of inserted probes for each JVM, the Probe Manager is also capable of automatically identifying all probes that have been previously inserted (whether statically or dynamically) by parsing and relatively quickly examining a method's bytecodes. This mechanism is also used to validate external configuration files to ensure that they accurately reflect the current set of inserted probes in a given instrumented version of the application.

3.3 Configuration Manager

While the act of placing probes is quite straightforward, it would become tedious if one had to redefine and redeploy probes each time ProbeMeister connected to the application -- for *each* remote component. For this reason we implemented a probe configuration manager. The Configuration Manager is responsible for tracking and recording all probe deployments to each application. The current configuration can be viewed and saved (to XML-based configuration files) at any point. Once saved, a configuration can again be viewed, and also reloaded and reapplied. Reapplying a configuration causes all probes to be reconstructed and then deployed to the selected application.

A second use of configuration files is to define probe sets that target specific activities or parts of the application (e.g. file access, network traffic, etc.). Using these sets, one could load and monitor the output from one probe set, then deapply the set (which removes deployed probes) and reapply another set.

3.4 User Interface

The graphical user interface (see Figure 2) provides access to all of the features described above. Virtual machines (applications) waiting to attach to ProbeMeister are announced at the bottom of the display. As stated, the user may also initiate a connection (using the menus) to a remotely running virtual machine. Once connected, the user *resumes* the virtual machine's execution. Each tab in the display represents a different remote virtual machine running a separate component or application. This figure shows two applications that ProbeMeister is connected to. The

[3] Siena is the event bus for many projects in the DASADA program.

[4] Probe event content may include probe location, invocation time, stack traces, user-assigned values, and method arguments, for example.

[5] We switched from using IBM's Jikes because of licensing constraints (only evaluation licenses were available), and because it was no longer being improved

[6] We found that it takes several minutes to define a new class which is a concern to us, but we have not yet studied this issue in detail.

first is a remotely running GeoWorlds[7] client application, the second is a service component used by the client. GeoWorlds is a central part of the software testbed within the DASADA project because it is a distributed component-based application that dynamically assembles itself on-demand.

The interface lists all of the application's classes that have been loaded (the core JDK classes have been filtered out using the controls at right). The *add()* method has been instrumented with a simple probe -- this probe outputs a user-provided string to the application's console. From the list of classes one can also see a class that ProbeMeister has dynamically deployed (called OBJS_Breakpointer) into the remotely running virtual machine. ProbeMeister automatically deploys this class into each attaching JVM to control breakpointing (methods belonging to classes in a remote JVM can only be invoked at breakpoints).

Probes are inserted by dragging a probe from the list of probes onto the desired method. Most probes require some configuring and present displays for customization. The probe icons are used differentiate between deployed and undeployed probes, and simple probes and probe stubs (described later).

The Gauge Deployment Requests list illustrates how external tools may suggest deployment locations within ProbeMeister. These tools may also automatically deploy probes without user intervention. This interface is only present when requested (and is the subject of a future paper on Software Surveyor).

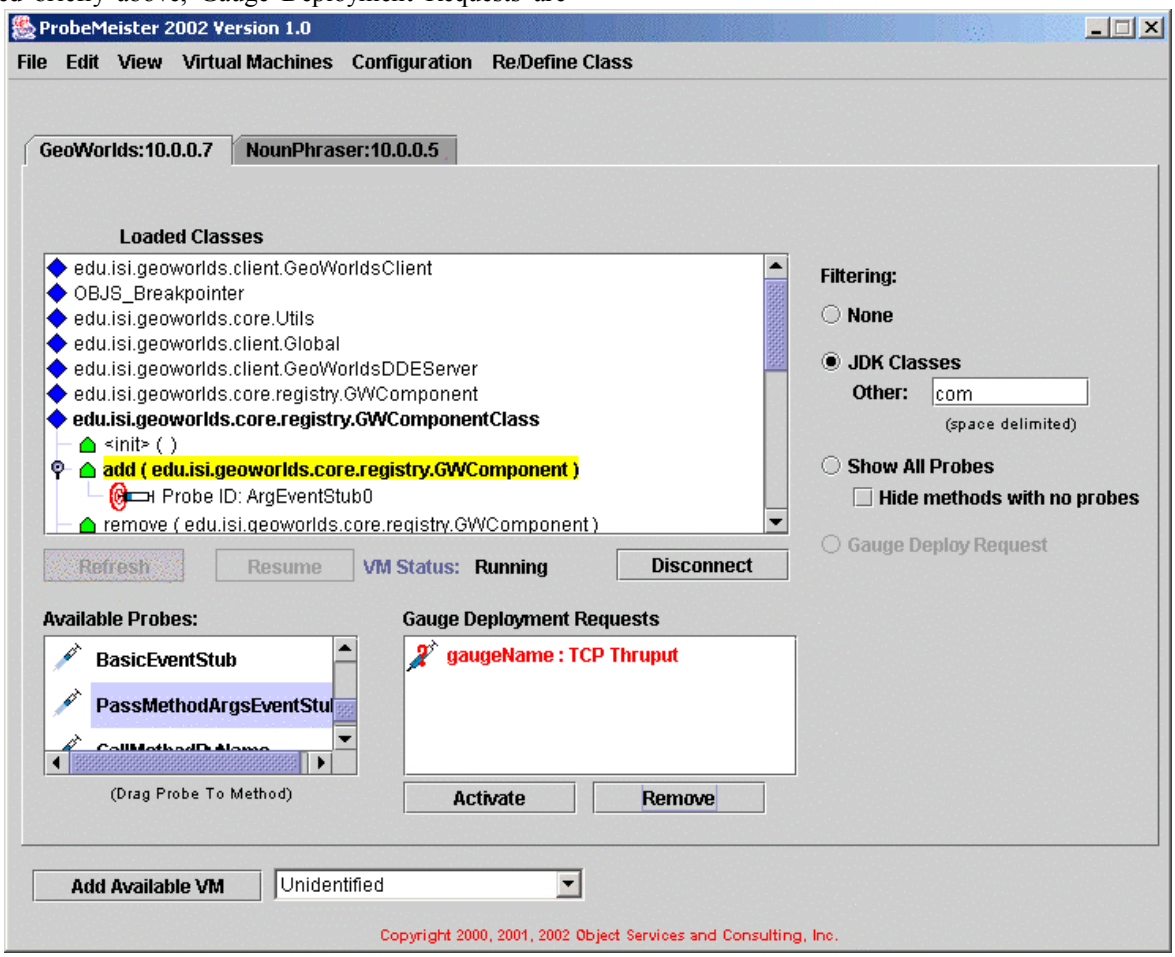
3.5 Other Interfaces

ProbeMeister provides access to its functionality through local and RMI-based programmatic interfaces. As seen in Figure 2 and discussed briefly above, Gauge Deployment Requests are sent over RMI to ProbeMeister. These external software gauges consume the events emitted (over the event bus) by deployed probes, so when gauges are first activated they suggest or auto-deploy (via deployment requests) the probes required to monitor the targeted activity.

4 Probes

ProbeMeister provides a Statement Factory to generate bytecode probe definitions on the fly. Probes are defined like *recipes* where the ingredients are Java bytecodes. Defining a probe recipe requires identifying the series of calls to be made to the Statement Factory. Each call adds one or more Java bytecodes. While several probe recipes are provided, others can be added to the library by extending the BytecodeProbeInterface. Probes can also be constructed in an ad hoc manner by directly calling the Statement Factory via the programmatic interface. Furthermore, the Statement Factory can also be extended with more functional bytecode building blocks. The following example illustrates how the simple PrintStringProbe class creates bytecode using the Statement Factory and inserts it into a specified method (defined in a bytecode location - `bLoc`).

```
[a] StatementList sList =
      BytecodeMgr.createStatementList(bLoc);
[b] StatementFactory.createPrintlnStmt(sList,
                                userStringToPrint);
[c] SimpleProbe simpleProbe = new SimpleProbe
            (probeID, probeDescription,
             probeType, sList, bLoc);
[d] BytecodeMgr.insertProbe(simpleProbe);
```



Initially [a], a new structure (StatementList) is created that will hold (and validate) the probe-specific bytecode. In [b], the call to the Statement Factory's `createPrintlnStmt()` generates bytecode that outputs the specified string, and then inserts the custom bytecode into the StatementList. In [c], a new simple probe wrapper is created (it knows how to deploy simple probes). It is passed a unique probe ID, a probe description, a probeType (PrintStringProbe), the StatementList, and the bytecode location. Finally, the probe is inserted into the targeted method. Once this is done, `redefineClasses()` may be called to propagate the update to the remote JVM.

ProbeMeister defines two types of deployable probes: simple probes and probe stubs. Simple probes are self-contained units of code. While they may call out to other methods owned by the application, they do not require any more probe-specific code to function. The current set of predefined simple probe recipes include a probe that outputs a user-defined string (discussed above), one that outputs the method's argument values, another that calls a specified static method, and a similar probe that calls a static method using introspection wrapped with exception handling. Simple probes may output information to the console of the remote application (such as argument values), or modify method state, for example. But, without supporting code, a probe cannot emit events. This is one motivation for probe stubs.

As there is only so much one can do with a probe in a single method, we found a need for a probe that could be divided in two: we call them probe stubs and probe plugs. A probe stub, like a simple probe, may perform intra-method manipulations such as modifying argument values or outputting data to the console. However, a probe stub is also able to perform more complex tasks because it calls out to one of an array of probe plugs. For example, two of our pre-defined stubs (probe recipes) include one that emits status information, the stack trace, a user-defined string, an event name and sub-event name; the other also emits the set of method arguments. This information can then be passed to a plug for further processing, and even return values back to the stub (e.g. to effect state changes).

Unlike probe stubs, which are written in Java bytecode, probe plugs can be written in Java. This really simplifies the writing of the bulk of the probe's functional code. A probe plug provides specific functionality that may perform any task. We currently use probe plugs to emit data from the probe stubs over the Siena event bus. Stubs are matched to plugs by their method signatures. When a user selects a probe stub to install, the Probe Manager returns a list of all compatible probe plugs from the ProbePlugCatalog. The user then selects an appropriate plug based upon its functional description. Like simple probes and probe stubs, new probe plugs can be added by registering them in the appropriate persistent catalog.

When stubs will be used, either the remote virtual machine must include the associated probe plug classes in its classpath, or ProbeMeister can port the probe plug classes to the remote virtual machine on the fly. The latter of course is preferable, as otherwise the plug code will need to be copied to each remote computer. However, if a considerable number of classes need to be deployed it may require significant overhead[7].

[7] The Siena Distributed Event Server is composed of 54 classes, making it more practical to copy the jar file to each site. However, it is likely that we could modify it to reduce the number of classes significantly, thus making it possible to deploy on the fly.

Finally, the Statement Factory validates the structure of each probe (only the Statement Factory can insert bytecode into a StatementList) and uses a wrapper mechanism to ensure that the probe can be removed once deployed.

5 Issues

There are a number of issues and limitations that are worth mentioning. First of all, as previously discussed, simple probe output is constrained to the remote JVM's console window because the probe code is executing within the context of the remote application. This is useful for certain types of debugging and monitoring, especially if the application is local. But, if the application is distributed, there must be a way to collect the probe output from each remote JVM. Using probe stubs and supporting code a probe can emit events external to the remote host. As previously mentioned, we currently support this capability using Siena. The events generated by the probes are published to a remote Siena event server and subscribed to by our user-oriented Siena event monitor (and other Software Surveyor gauge tools), which then displays the event data in a web browser. Other event publication schemes are also possible. For example, one could use the Java JDK 1.4 Logger API to emit probe events in the form of log messages via TCP streams to a remote collection system.

Another issue is probe control. Currently probes deployed in the remote application can only be disabled by removing them. One potential alternative would be to simply modify the probe bytecode by inserting a jump instruction to bypass the probe code. This is slightly more efficient than removing the entire probe and reinserting it at a later time. Another alternative would be to port a new class that contains a vector of Boolean switches. Each probe would then check its own on/off value in this vector prior to executing. ProbeMeister would modify the values in this vector by remotely invoking a method to alter the on/off values. However, (unlike method modification) object invocations using the JDI API require that the remote application be at a breakpoint. We have yet to measure the overall cost of this approach. Although, given that remote probe removal is on the order of 250 milliseconds it has yet to become a major issue.

While using the JDI API, we've noticed three important constraints. First, to modify a method ProbeMeister needs a copy of the complete bytecode of the class because critical pieces found in a .class file are not defined at the method level. This includes, for example, the bytecode boundaries in which a given attribute is valid, as well as the definition of exception handlers. Unfortunately, we have learned that the JVM cannot synthesize class definitions, so at this time ProbeMeister must have access to copies of all of the bytecode it may edit. Second, there is no straightforward method to reliably cause a breakpoint to occur in the remote JVM. While one can arbitrary set a breakpoint using the JDI interface, the problem is knowing *where* to set the breakpoint. We have created a simple mechanism that allows ProbeMeister to cause a breakpoint at anytime (using our Breakpointer class as described earlier), but only if the application attaches to ProbeMeister at startup (because we know where the application will *begin* execution!). We have not yet looked for a reliable way to port the Breakpointer class to the targeted application if ProbeMeister attaches to a running application. However, ProbeMeister needs to set breakpoints so it can invoke methods on remote objects.

The final constraint is that when an application connects to ProbeMeister there is no way to identify it. We have implemented a mechanism that will read special ProbeMeister-specific parameters that can be included in the command line (this requires ProbeMeister to invoke methods in the remote JVM to access these values). Preferably, such metadata would be made accessible via the JDI API prior to accepting a connection.

Another limitation is that our supplied probes cannot modify a method's arguments when the symbol table is not included in the class (a compile-time option can strip a class of its symbol table). However, a probe could modify these values by cross-referencing the original source, though we have not tried this. Not having the symbol table limits what a probe can do in a running application, for better or worse. Still, if needed, it is possible to access a method's local variables by statically instrumenting the source code. For example, we have instrumented the source code of some core JDK classes (e.g. java.io.File and java.net.URL) with special probes that provide access to more details than otherwise currently possible with our bytecode probes.

With respect to performance issues, we have noticed that while probe deployment is relatively quick, remotely deploying new classes appears quite costly – on the order of 100+ seconds. We have yet to investigate this issue to determine the source of the problem, but we did notice significant bandwidth usage.

Like any other code writing, it is important to extensively test new probes as poorly written probes can easily cause catastrophic effects (the creation of the Statement Factory was intended to minimize such problems). And while the inclusion of exception handling in a probe addresses some of these concerns, it is still quite easy to write damaging code if one is not careful.

6 Plans

We are working to extend and enhance ProbeMeister. As mentioned earlier, probes need a distribution infrastructure to emit events. As the Java JDK 1.4 Logger can send logged data to a remote location, this will be a lightweight alternative to using Siena. If the application is already using this mechanism, then we could also merge and remotely route application output and probe output together. Furthermore, the Logger API defines logging levels that we plan to extend to control which probes emit events. We plan to explore this approach to turning on and off probes, in addition to the current "deploy, remove, and redeploy" approach.

Another feature we are exploring is to remove the limitation requiring local bytecode access so that a method can be modified, and probe installed. This requires that ProbeMeister have access to a copy of every classfile in which a probe might be deployed. To alleviate this, we plan to deploy helper classes into the remote JVMs that will load and transmit (back to ProbeMeister) the classfiles to be modified. This will also guarantee that the classfile used by the application is the same version that ProbeMeister is modifying.

Currently, ProbeMeister is limited to blind instrumentation. That is, it does not display the source code, or allow the user to specify probe location as a source code line offset. We plan to extend our user interface to support the ability to specify the location of a probe similar to how breakpoints are placed within a debugger interface.

Finally, we plan to define some default probe configurations for addressing common monitoring needs, such as network activity, binding failures, and file access. This would allow a user to quickly isolate certain types of problems, after which they could manually deploy probes into specific components given what they had observed.

7 Acknowledgements

Many thanks to Sun Microsystems Java CAP team for providing access to, and support of, JDK 1.4 (special thanks to Jim Holmlund for his responsive support in debugging JDI-related issues). Thanks also to the reviewers for their invaluable feedback.

8 Related Work

This is a partial list of related Java-specific tools.

Bytecode Modifiers

- JOIE: The Java Object Instrumentation Environment , http://www.cs.duke.edu/ari/joie/
  - Geoff Cohen (Duke/IBM), Jeff Chase (Duke), and David Kaminsky (IBM), Automatic Program Transformation with JOIE in Proceedings of the 1998 USENIX Annual Technical Symposium
- CFParse , http://www.alphaworks.ibm.com/
- BIT: Bytecode Instrumenting Tool , http://www.cs.colorado.edu/~hanlee/BIT/index.html
- Jikes Bytecode Toolkit , http://www.alphaworks.ibm.com/tech/jikesbt
- Bytecode Engineering Library , http://jakarta.apache.org/bcel/

Commercial Probe Deployment Tools

- JProbe Java Performance Tools http://www.klgroup.com/jprobe/
- JTrek , http://www.digital.com/java/download/jtrek/index.html
- NuMega DevPartner® Java™ Edition , http://numega.com
- RootCause -Java and C++, http://www.ocsystems.com

Research Probe Deployment Tools

- NTWrappers - C++ - http://www.teknowledge.com

Pre-instrumented JVMs

- Jinsight , http://www.alphaworks.ibm.com/tech/jinsight
- eTective , http://www.averstar.com/products/etective.html
- Binary Component Adaptation for Java (BCA), http://www.cs.ucsb.edu/oocsb/bca/index.html

9 References

[1] Software Surveyor Project, http://www.objs.com/DASADA/index.html

[2] DARPA DASADA Program, http://www.darpa.mil/ito/research/dasada/projects.html

[3] D Wells and P Pazandak, *"Taming Cyber Incognito: Surveying Dynamic / Reconfigurable Software Landscapes"*, In Proc of 1st Working Conference on Complex and Dynamic Systems Architectures, Dec 12-14, 2001, Brisbane, Australia.

[4] Sun Microsystems JDK 1.4 Java Platform Debugger Architecture, http://java.sun.com/j2se/1.4/docs/guide/jpda/jdi/index.html

[5] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf **"Design and Evaluation of a Wide-Area Event Notification Service"**. *ACM Transactions on Computer Systems*, 19(3):332-383, Aug 2001

[6] Bytecode Engineering Library, http://jakarta.apache.org/bcel/

[7] *M Coutinho, R Neches,et al*, **GeoWorlds: A Geographically Based Information System for Situation Understanding and Management**, In Proc of 1st Intl Workshop on TeleGeoProcessing, May 6-7, 1999, Lyon, France.