DAVID JOSEPHSEN

iVoyeur: opaque brews

David Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.

dave-usenix@skeptech.org

THE JAVA SERVLET CONTAINER MODEL is one of the most popular ways to provide dynamic content to a Web browser. If you haven't had the pleasure of dealing with one, a "Servlet Container" is what you get when you combine a small Web server with a Java program called a servlet. The job of the Web server is simply to accept HTTP requests from the network. The Web server can do things such as parse host names from HTTP headers and perform SSL handshakes as any normal Web server might.

But instead of handling the HTTP requests itself, this Web server passes them to a program called a servlet. The servlet then generates some content, usually in HTML, for the Web server to pass back to its client. The servlet itself is not a binary program but, rather, Java bytecode. It runs inside a Java Virtual Machine (JVM), which provides a runtime environment complete with threading, memory management, much-hyped security, interfaces to other Java and system functions, and presumably everything else a servlet could want. There are quite a few implementations of this model, including Apache Tomcat, BEA Weblogic, IBM WebSphere, and Oracle Application Server, but the basic idea is the same.

In practice it works pretty well until it doesn't, and the complexity introduced by the model makes it nearly as unpopular among system administrators as it is loved by Web developers and their managers. The problem from the systems perspective is the virtual machine. Because the servlet we are trying to troubleshoot or monitor is running inside a virtual machine, our system tools are rendered useless. From the outside, all we can see is the JVM process itself.

If you have a single Web site running on the JVM, then the memory footprint of the JVM is pretty close to the memory footprint of your Web site. Likewise, the CPU load induced by the JVM is pretty close to the CPU load of your Web site. But what if you have five Web sites being vhosted on your Web server? Now a single JVM is running five servlets. Which of them is the one hogging your CPU? Even if you don't have a problem to diagnose, you at least have a capacity planning conundrum, but, believe me, it gets worse.

Let's say you do have a problem and it isn't resource-bound. For example, the application is hanging, or it is crashing outright at unpredictable times. Even with a single servlet in the JVM, tools such as strace, dtrace, and systemtap can only be of limited value, because the JVM has its own internal memory management and thread model. All you can see from the outside is what the JVM's doing, and since it allocates most of the resources it needs up front (including, e.g., its database connection pools), that usually isn't very much.

And speaking of hanging and crashing, I've been privileged in my short career as a Tomcat administrator to see the JVM crash in all sorts of intricate, fascinating, and unpredictable ways. So I can say from personal experience that the servlet container model makes for interesting monitoring fodder in that, short of directly parsing the HTML it returns, it can be difficult to even define a criteria for "functional" that actually describes a functional servlet container. So if you need to monitor specific metrics on your applications, or you've ever wondered just what the heck is happening inside that virtual machine in general, this article will provide some tips, from a systems perspective, for penetrating the black box that is the servlet container.

Profiling

Like their non-Java counterparts, JVM system profilers can provide detailed info on what the JVM is spending its time doing. There are two primary ways of accomplishing this. The first is by registering for messages from built-in instrumentation libraries such as the Java Virtual Machine Tool Interface [1]. The second is by using a process called byte-code injection, wherein known byte-code instructions are detected and preempted with snippets of management-related code. Byte-code injection is more accurate but is much more expensive. Many profilers exist, but most assume that you are a developer operating within an IDE such as Eclipse. For a sysadmin trying to debug a problem on a production system, you can't do much better than hprof [2].

The tool hprof is a simple, powerful JVM profiler that's been included in the JDK since version 5.0. There is nothing to install, and there are no dependencies (well, other than the JVM itself). Simply enable it by passing a -X switch to your JVM options. If you're using Tomcat, for example, you would simply need to add a line similar to:

```
-Xrunhprof[:options]
```

to your startup.sh or Tomcat init file. When the program exits, or whenever the JVM catches a sigquit (kill -3 in UNIX), hprof writes its profiling information to a standard text file. It can provide a stack trace of every live thread in the JVM, heap profiling (what's using all the memory?), and CPU profiling (what's using all the CPU?). It can use either byte-code injection or the Java Virtual Machine Tool Interface, but in practice the former method incurs such a heavy performance penalty that it is, in my experience, unusable for troubleshooting in production environments.

By way of an example, we recently had a problem with several applications hanging on a production Tomcat system. The application hang was accompanied by a CPU spike, so we used hprof to obtain some CPU samples and clicked around the site until the problem showed up. The CPU profile (near the bottom of the hprof dump) looked something like this:

```
CPU SAMPLES BEGIN (total = 206138) Tue Jul 24 22:00:30 2007
 rank      self       accum      count      trace method
   1    42.02%    42.02%       180          481612
oracle.jdbc.driver.OracleDriver$1.<init>
   2    14.94%    14.94%     30796          478299
java.net.SocketInputStream.socketRead0
   3     1.09%    11.09%     22868          495714
java.net.SocketOutputStream.socketWrite0
   4     3.94%     3.94%      8116          495716
java.net.SocketOutputStream.socketWrite0
<snip>
```

The number 1 user of the CPU (at 42%) was the jdbc driver, the glue between our Java application and its Oracle database backend. The number listed under the "trace" column is a unique ID with which we can locate and examine the stack trace of this thread. Toward the top of the hprof dump is the stack trace in question:

```
TRACE 481612:

oracle.jdbc.driver.OracleDriver$1.<init>(OracleDriver.java:1425)
oracle.jdbc.driver.OracleDriver.getSystemProperty(OracleDriver.java:1423)
oracle.jdbc.driver.OracleDriver.connect(OracleDriver.java:840)
        <snip>

oracle.jdbc.pool.OracleDataSource.getConnection(OracleDataSource.java:165)
  pkg.dbgCalls.getConnection(dbgCalls.java:294)
  pkg.dbgCalls.getTrackingId(dbgCalls.java:1843)

  org.apache.jsp.index_jsp._jspService(index_jsp.java:145
```

The last line in the stack trace lists the file and, more specifically, the line number in the file that contains the code that is hogging our CPU. This problem ended up being caused by an ancient copy of the ojdbc14.jar in the WEB-INF folder of the application. Not all problems are this cut-and-dried, and the CPU sampling technique becomes less useful the longer the JVM operates (which makes troubleshooting problems you can't reliably replicate difficult). Also, hprof can't really provide real-time analysis, and it doesn't lend itself to ongoing performance or availability monitoring. Generally, however, hprof is great at providing really specific information like this on demand with a manageable overhead, without raising the vulnerability footprint of the server, and without installing additional software.

JMX

The JVM itself contains instrumentation code for monitoring and management and an API for accessing monitoring and management information in the form of Java Management Extensions (JMX) [3]. JMX is composed of a service that brokers monitoring and management requests to the JVM (called an "mbeans server") and several connectors, which expose the API in various forms such as SNMP and RMI (a Java protocol used by JMX-based monitoring apps). In-house developers can also use the JMX libraries to instrument their applications directly; however, the JVM's instrumentation is sufficient for most monitoring purposes.

If you didn't parse much from that last paragraph, I can sympathize. Java documentation often reminds me of a Frank Herbert book (which is to say, overly concerned with jargon), so I'll attempt another explanation (this time in English). JMX is a direct answer to the problems stated in the opening paragraphs of this article. It provides a window into the operation of the

JVM, and it makes possible such things as a JVM equivalent of the UNIX top program and much more. Further, the same performance and monitoring data can be consumed several different ways, including SNMP and RMI (which makes it possible for a top-like program to monitor a server remotely). When a monitoring vendor says that its app has "Java Integration," that probably means they have a home-brew built-in JMX agent of some sort.

In fact, a JVM equivalent of top called Jtop is included with the current JDK in the demo/management subdirectory. Jtop can give real-time data about the JVM's CPU utilization on a per-thread basis. To use it, you must first enable JMX in your JVM by passing a couple of command-line switches to your startup script. Enabling RMI (remote connections) access to your JVM is a dangerous thing to do. JMX exposes sensitive configuration info such as user names and passwords, as well as the ability to change just about anything related to the operation of the JVM. So you should follow Sun's instructions [4] for enabling authentication and SSL on your JVM's RMI connector. The quick, dirty, and highly insecure way to enable JMX is:

```
-Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.port=1223 \
-Dcom.sun.management.jmxremote.ssl=false \
-Dcom.sun.management.jmxremote.authenticate=false \
```

Those switches aren't specific to Jtop; any RMI-based JMX app will require them. Jtop is fairly nifty, but it doesn't even hint at the amount of information that JMX exposes about a running JVM. Jconsole [5], however, included with the JDK and located in JDK_HOME/bin/jconsole, is a fully functional graphical frontend to the JVM. It provides real-time in-depth analysis of a running JVM's memory allocation, CPU utilization, run-time parameters, and stack traces. Jconsole can do everything from providing the exact CPU and memory utilization of a single thread to adding and deleting accounts from the tomcat-users.xml. You do incur quite a bit of overhead using it, but not enough to obviously affect the response time of a running application.

There are several ways to get this info out of JMX and into your monitoring system. The easiest is perhaps JMX's SNMP connector [6]. Simply set the system property for it in the JVM like so:

```
-Dcom.sun.management.snmp.port=portNum
```

Then create an ACL file in JRE_HOME/lib/management/snmp.acl (there's a sample file: JRE_HOME/lib/management/snmp.acl.template). Once this is done, the entirety of JMX is exposed via SNMP for your favorite snmp-enabled monitoring system to make use of. You may even define traps for trap collectors and passive monitoring systems.

For Nagios users, there is a check_jmx [7] plug-in that can query the status of any JMX metric that you see in jconsole. Its syntax is typical of Nagios plug-ins in general. For example, if you were interested in monitoring the JVM's heap you could use:

```
./check_jmx -U \
service:jmx:rmi:///jndi/rmi://myHost:1223/jmxrmi -O \
java.lang:type=Memory -A HeapMemoryUsage -K used -I HeapMemoryUsage -J \
used -vvvv -w 1000000000 -c 1500000000
```

Finally, if you're thinking about rolling your own RMI-based monitoring script, Sun's JMX tutorial [8] has plenty of sample code to get you started, and there's a great white paper on the subject called "JMX Interoperation with non-Java Technologies" [9] on Daniel Fuchs's blog.

JMX is not a panacea. It's still dependent on the JVM itself operating correctly, and that's a pretty big dependency. It does, however, provide excellent insight into the functioning of an operational JVM, and it's allowed me to nip a few problems in the bud before they had the opportunity to destabilize the JVM, and hey, any visibility is a net gain IMHO.

Take it easy.

REFERENCES

[1] http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html.

[2] http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.

[3] http://java.sun.com/j2se/1.5.0/docs/guide/management/overview.html.

[4] http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html.

[5] http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html.

[6] http://java.sun.com/j2se/1.5.0/docs/guide/management/SNMP.html.

[7] http://www.nagiosexchange.org/Misc.54.0.html?&tx_netnagext_pi1%5Bp_view%5D=808&tx_netnagext_pi1%5Bpage%5D=20%3A10.

[8] http://java.sun.com/j2se/1.5.0/docs/guide/jmx/tutorial/tutorialTOC.html.

[9] http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/JSR262_Interop.pdf.