Managing Java Application Performance

How to Proactively Discover, Diagnose and Resolve Performance Issues in Java Applications

A Technical White Paper

www.eginnovations.com

Total Performance Visibility

Introduction

Java-based applications are powering many business-critical IT services today. Uses of Java technologies span several domains including healthcare, logistics, finance, insurance, education and many more. In all of these instances, Java technology is the middleware in which the business logic resides. Many production installations also include home-grown application components, running on standard Java application servers such as Oracle WebLogic, IBM WebSphere, SAP NetWeaver, Apache Tomcat and JBoss, to name a few.

Because it hosts the core business logic, the performance of the Java middleware tier has a significant impact on the performance of the business services that it supports. Since all Java applications (whether client-server or web-based) run on a Java Virtual Machine (JVM), monitoring of the JVM can provide key insights into performance issues that could have a significant impact on the supported business services. For example, a single run-away thread in the JVM could take up significant CPU resources, slowing down performance for the entire service. Alternatively, a deadlock between two key threads could bring the business service to a grinding halt.

Additionally, the functioning of the web containers/application servers, on which the application components are hosted, also affects application performance. Moving up the stack, the application logic and transaction flows between application components, method calls, third party API calls, and database queries - to name a few - can all affect Java application performance as well.

This white paper focuses on monitoring at the JVM layer and provides use cases to highlight how JVM monitoring can help pinpoint the cause of complex performance issues. Web container monitoring and transaction monitoring within and across JVMs are also important but are beyond the scope of this whitepaper.

eG Enterprise, from eG Innovations, offers total performance visibility for Java applications, and this whitepaper shows how built-in JVM monitoring capabilities in eG Enterprise enable Java application performance issues to be diagnosed and resolved quickly, thereby ensuring peak performance with low downtime.

Importance of the Java Virtual Machine

The core of the Java middleware tier is the Java Virtual Machine (JVM). It is a main component of the Java architecture and is part of the Runtime Environment (JRE) - see *Figure 1*. JVM is the platform on which applications function and the Java code is executed. The JVM is considered as virtual machine because it provides an interface that does not depend on the underlying operating system and machine hardware – which helps deliver on Java's unique benefit of writing code once and running it anywhere.

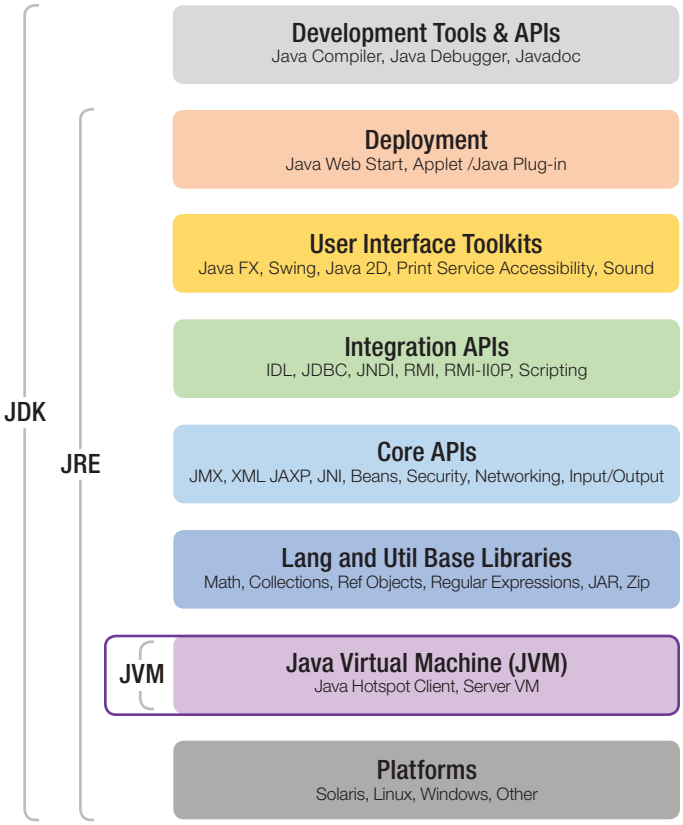

Figure 1: Architecture of Java Platform Standard Edition (JVM is part of Java Runtime Environment)

How the JVM Works

The Java compiler does not produce native executable code. Instead, it creates what is called as byte code (see *Figure 2*). Byte code is a highly optimized, platform-independent set of instructions designed to be executed by the JVM. The JVM acts as an interpreter that reads and understands byte code and executes the corresponding native machine instructions.

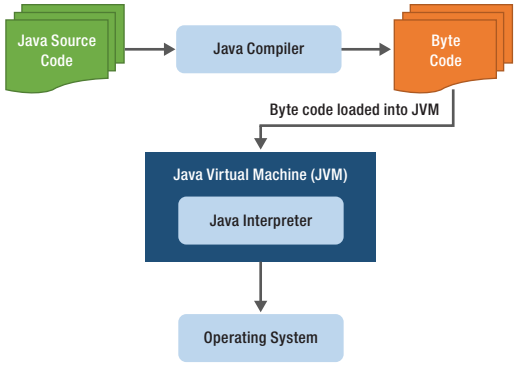

Figure 2: How the JVM works

Key functions of a JVM

- Loading bytecodes from application class files
- Verifying the loaded byte codes
- Linking the program with the necessary libraries
- Allocating memory needed by the Java program
- Managing calls between the program and the host environment
- Management of heap and non-heap memory and garbage collection
- Handling threads their creation, execution and synchronization

In the past, custom byte-code instrumentation had to be developed to look at the internals of the JVM. Newer JVMs (JVM 1.5 or higher) come with extensive pre-built instrumentation capabilities that monitoring tools can tap into. The Java Management eXtension (JMX) interface of the JVMs can be used for this purpose. The monitoring can be done in an agent-based or agentless monitoring – i.e., using an agent installed on the system, or using one that resides outside the server hosting the target JVMs. *Figure* 3, below, highlights the metrics and details that can be collected regarding the performance of the JVM.

JVM Resource Usage

- Heap memory usage, Eden Space, Survivor Space, Tenured Gen
- Non-heap memory usage, Pool Code Cache, Perm Gen
- CPU utilization of JVM

JVM Threads

- Blocked threads
- Waiting threads
- High CPU threads
- Peak threads
- Daemon threads
- Deadlock threads

Java Classes

- Loaded classes
- Unloaded classes
- Total classes

Garbage Collection

- Number of garbage collection operations
- Time taken for garbage collection
- % of time spent by JVM for garbage collection

JMX Connection to JVM

- JMX availability
- JMX response time
- Any changes to PID?

Visibility into the Server Operating System

- Incoming and outgoing traffic to the service
- Server uptime, disk activity, CPU and memory statistics

Figure 3: Key metrics regarding the performance of a JVM and its underlying operating system

By monitoring these JVM metrics, IT Ops and DevOps personnel can get answers to critical performance questions:

- What is the average CPU utilization of the JVM? Which threads are responsible for the CPU usage and what line of code is each thread executing?
- How many threads are running in the JVM at any time? Is there any leakage of threads (i.e., are threads being started and left running over time)?
- Are there any deadlocks happening in the JVM? Which threads are responsible for the deadlock, and which lines of code (which modules, classes, files) were they executing prior to the deadlock?
- Are there any blocked threads? Which threads are blocking them and which lines of code are responsible for this?
- How efficient is the JVM's garbage collection process? How long does garbage collection take?
- What is the heap usage of the JVM and which object types are responsible for the memory usage of the JVM?
- When was the JVM last restarted? What is the current uptime of the JVM?

The following sections illustrate examples of how comprehensive monitoring of the JVM can help easily identify and quickly diagnose the root-cause of two common categories of performance problems that impact Java-based business services. In the first example, we will consider a runaway Java application, one that suddenly starts taking up a lot of CPU resources and we will highlight how the exact thread and line of code responsible for this can be identified. In the second example, we will consider a hung Java application and depict how JVM monitoring can pinpoint the code snippet that is responsible for a deadlock.

Scenario #1: Diagnosing a Run-away Java Application

Consider an online web store named *zapstore* that uses Java technologies. *Figure 4* depicts the monitoring dash-board for this business service. A color code next to the service name denotes its current status. While the color green denotes normalcy, different shades of red indicate problems of different levels of severity.

Figure 4 illustrates that the *zapstore* service is having a critical problem. IT managers can click on a service to get details of the health of each of the tiers supporting this ser- vice. The first drilldown highlights the user experience – i.e., are users happy and productive, or are they unsatisfied? User experience can be monitored using a synthetic approach, by recording typical user interactions with the service and replaying the interactions to measure availability and response times. User experience can also be monitored passively, – observing how real users are accessing the service and how the service responds to their requests.

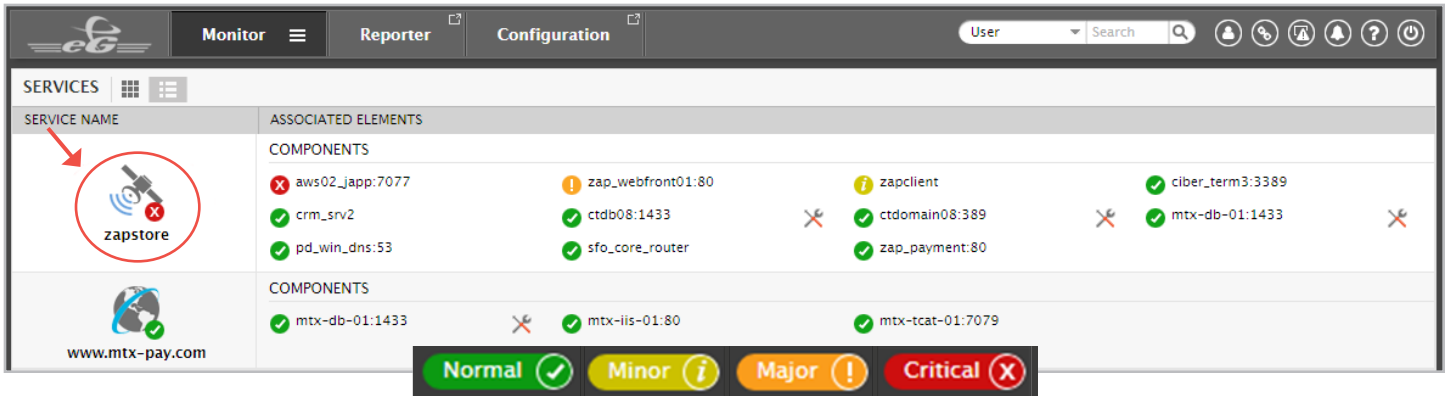

Figure 4: eG Enterprise's Business Service Dashboard identifying a critical problem in zapstore service

Figure 5 displays the user experience measured passively using a plug-in installed on the web servers. Here, we see that there are three slow transactions. One of them - the *Update_Cart* transaction - is taking over 19 seconds to execute. Normally, a poor response time problem would be a critical issue as it is business-impacting. But, the lower priority alert on the console indicates that the monitoring system has intelligently determined that the response time slowness is an effect of a problem and not the cause.

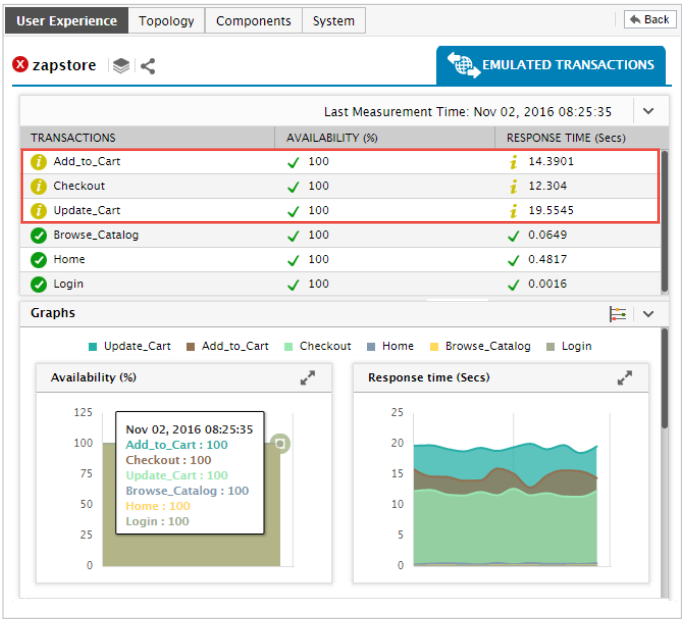

Figure 5: Monitoring transactions to zapstore service

To understand where the real problem lies, an administrator can click on any of the transactions in *Figure 5*. The drill-down reveals the topology of the *zapstore* service (see *Figure 6*). The service topology shows all the network devices and applications that are involved in service delivery and the inter-dependencies between them. The *zapstore* service uses a typical multi-tier architecture: users can access the service from a browser client, or by logging into a Windows terminal server from a thin client and then using a browser to connect to the service.

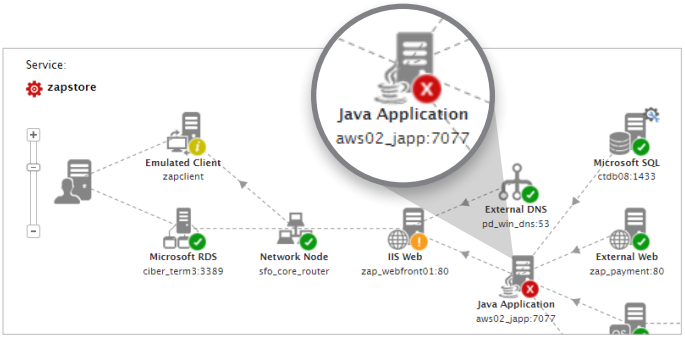

Figure 6: Topology map of the zapstore service

A Microsoft IIS web server front-end handles all the static content. The business logic is handled by the Java application *aws02_japp*. This Java application relies on an Active Directory server for user authentication and access rights validation. For access to the inventory database, the Java application queries an SQL database server. Customer support queries are directed by the Java application to a Windows-based CRM system. Payment processing is handled by an external web-based payment system.

Automated Root Cause Diagnosis

From *Figure 6*, we can see that the Java application, the IIS web server, the emulated client with slow transactions, all show problems. The color codes on the different tiers highlights the cause-effect relationships. By analyzing the dependencies between the different tiers, the monitoring solution has identified that the Java application is the probable root cause and the problems with the other tiers are probably its effects.

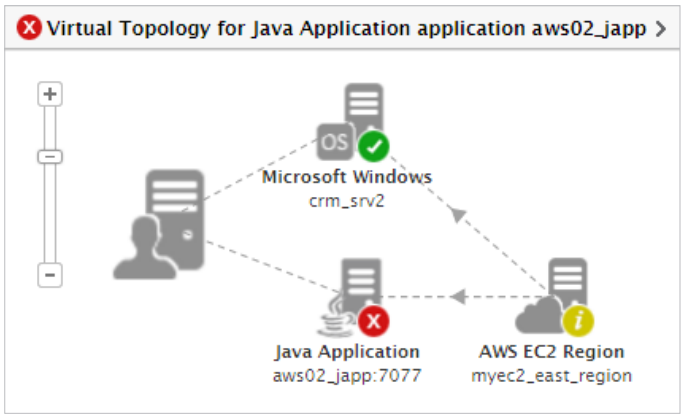

Figure 7: Virtual topology map shows the problematic Java application is hosted on Amazon EC2

Clicking on the problematic Java application in the service topology map takes the administrator to the next screen (see *Figure 7*). In *Figure 6*, we did not see the underlying infrastructure supporting the different application tiers. *Figure 7* shows the cloud infrastructure that is supporting the Java application server. The virtual topology shown here reveals that the CRM system and the Java application server are running on cloud virtual machine instances hosted on Amazon AWS EC2. This topology helps the administrator answer the question as to whether the problem is due to the Java application, or if it is because of the cloud platform on which the application is hosted. In this case, we can see that the problem originates in the Java application tier (displayed in red – critical state).

Drill Down to View Performance Metrics

Click on the problematic Java application in *Figure 7* to view its performance metrics. *Figure 8* shows the layer model view of the Java application. This layer model captures the key layers involved in the functioning of the Java application. All the metrics collected are mapped to these functional layers and the state of a layer is determined based on the state of all the metrics mapped to it (i.e., if all the metrics are assessed to be normal, the layer is in a normal state). The layers are hierarchically aligned, so performance issues at the lower layers are correlated with those at the higher layers (for example, in *Figure 8*, the *JVM Internals* layer has a critical alert, so the *JVM Engine* layer that depends on it is downgraded in priority, because a problem in the layer below can affect the layer above). The organization of metrics into layers ensures that administrators need not sift through hundreds of metrics to determine what problems exist in the infrastructure.

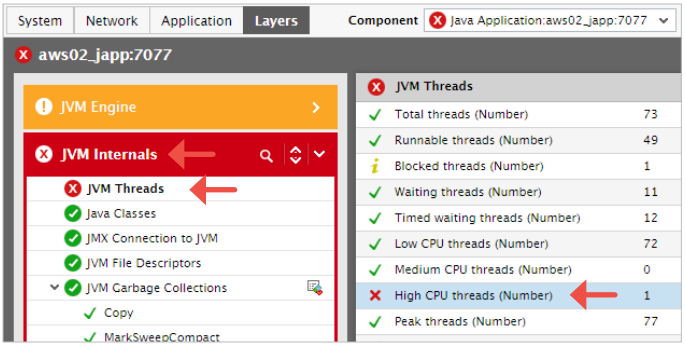

Figure 8: Layer Model Dashboard for viewing performance metrics of the Java application

The lower layers in *Figure 8*, namely, the Operating System, Network, and TCP are generic for all server applications, while the top two layers (JVM Internals and JVM Engine) are unique to the Java application tier.

- The JVM Engine layer monitors the CPU activity and memory usage inside the JVM.
- The JVM Internals layer looks deeper into the JVM and tracks the Java threads, classes loaded, and garbage collection activity.

In this example, looking at the layer model of the Java application server (see *Figure 8*), we can see that *JVM Internals* is showing a critical problem (indicated by the dark red color). On the left panel, the model shows that the problem pertains to *JVM Threads*. The panel on the right indicates that there is a high CPU thread that is causing the problem. Clicking on the magnifying glass icon next to the *High CPU threads* lets you drill down further (see *Figure 9*) to see the actual cause of the problem. Without the right tools, determining which thread and which line of code is causing the CPU spike would have been a very time-consuming exercise.

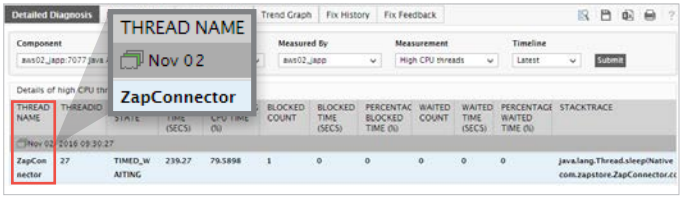

Figure 9: Detailed diagnosis isolating the Java thread with high CPU usage

From *Figure 9*, it is clear that the high CPU thread named *ZapConnector* is taking about 80% of CPU. Furthermore, the stack trace (see *Figure 10*) shows exactly which line of code the thread was executing when this high CPU state was last detected.

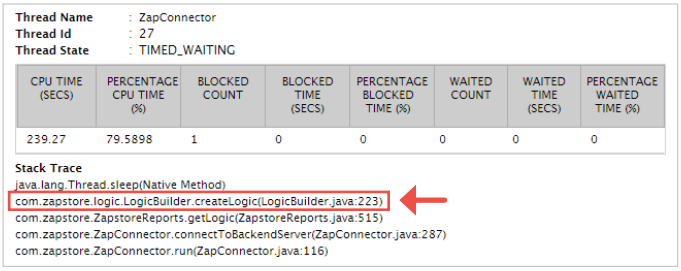

Figure 10: Stack trace for thread with high CPU usage

In viewing the stack trace, the most recently executed line of code is the one on the top. So in this case, it is line 223 of the *LogicBuilder.java* source file, which contains the *createLogic* method of the *com.zapstore.logic.LogicBuilder* class.

To see why there was a problem at this line of code, it is necessary to look at the *LogicBuilder.java* file in a text editor (see *Figure 11*). Here, we can see that line 223 of the source code is within a *while* loop. This code is supposed to loop one and a half million times and then sleep waiting for the count to decrease. Instead, the value of count is incorrectly reset to 0 at line 222, and this is causing the *while* loop to execute forever, thereby resulting in one of the threads in the JVM taking a lot of CPU. Deleting the code at line 222 would solve this problem.

```
213  public void createLogic()
214  {
215          long count = 0L;
216          while(!finish)
217          {
218                  try
219                  {
220                        if(count > 1500000)
221                        {
222                              count = 0;
223                              Thread.sleep(1);
224                         }
225                         count ++;
226                  }
227                  catch(Exception ex)
228                  {
229                      ex.printStackTrace();
230                  }
231              }
232          }
233     }
```

Figure 11: Exploring problematic code using text editor

This example has illustrated how a monitoring solution like eG Enterprise with end-to-end visibility, deep diagnosis and automatic correlation capabilities is able to identify the root cause of the problem—a runaway high CPU thread—in a multi-tier Web application within a few clicks and point to the exact line of code that is causing the issue.

During normal operation, administrators do not have to click through several screens to find the root cause of a problem. The eG Enterprise alarm window (see *Figure 12*) which comes up as soon as a user logs in, directly prioritizes alarms based on their severity. The alarms can be emailed to administrators, sent as SNMP traps and forwarded to other network management systems, or they can be used to open or close trouble tickets directly in helpdesk systems.

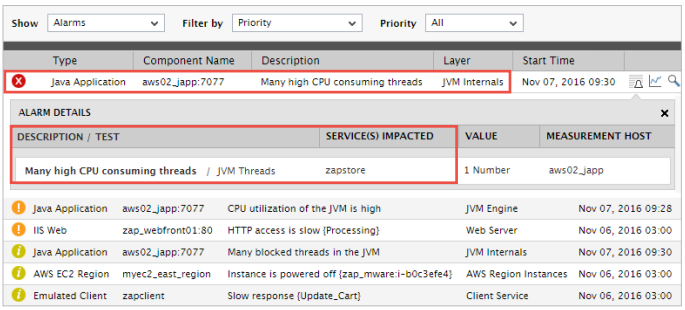

Figure 12: Alarms window in eG Enterprise displays alarms by severity (Most critical at the top)

Scenario #2: Diagnosing a Hung Java Application

The second performance problem scenario involves unexpected thread blocking. *Figure 8* from earlier, the Layer Model Dashboard used for performance monitoring of the Java application, showed another alert in *JVM Threads*: a blocked thread in the JVM (See *Figure 13* below). Let's investigate what is causing this.

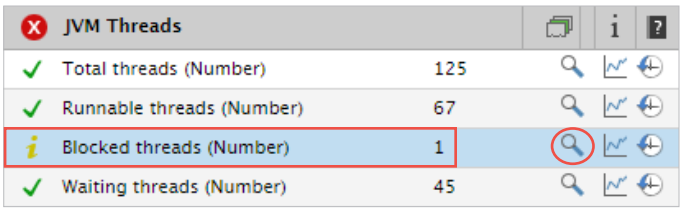

Figure 13: Layer Model Dashboard for the Java applications indicating blocked threads

Clicking on the diagnosis icon next to the *Blocked threads* metric brings up a diagnosis window showing all blocked threads.

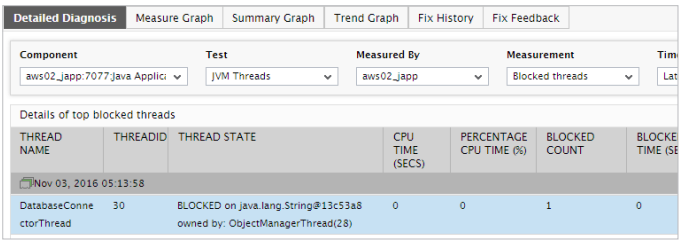

Figure 14: Detailed Diagnosis to find the blocked threads

Here we can see there is one blocked thread named *DatabaseConnectorThread*. Drilling down into the stack trace helps troubleshoot the problem further.

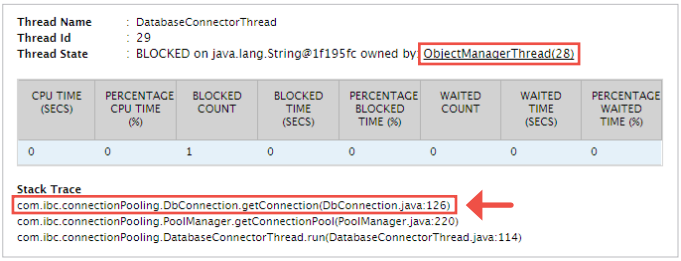

Figure 15: Stack trace for the blocked thread – DatabaseConnectorThread

Looking at the top of the stack trace, we can see where the *DatabaseConnectorThread* thread is blocked and at which line of code (126). the 'ThreadState' field clearly shows that the thread causing the blocking is the *ObjectManagerThread*.

Using a text editor to look at the *DbConnection.java* source file shows that the *while* loop in this class is inside a synchronized block (see Figure 16). The object used to synchronize the access to this block is a variable named *sync*. Looking at the variable declarations at the top of the source code (see *Figure 17*) reveals that the *sync* variable refers to the static string *test*.

```
122  public void getConnection()
123  {
124         synchronized(sync)
125         {
126            long 1 = 0L;
127            while (!finish1)
128            {
129                try
130                {
131                    Thread.sleep(3600);
132                }
133                catch(Exception ex);
134                {
135                    ex.printStackTrace();
136                }
137             }
138          }
139     }
140  }
```

Figure 16: DbConnection.java source file

```
1    package com.ibc.connectionPooling;
2
3    import com.ibc.objectPooling.*;
4    import java.util.Date;
5
6    public class DbConnection
7    {
8       public static boolean finish1 = false;
9
10      public static String sync = "test";
```

Figure 17: DbConnection.java variable declarations

Three key things can be uncovered by looking at the stack trace and the Java code source:

1. *DatabaseConnectorThread* is blocked and entering a synchronized block
2. Synchronization is controlled by a static string test
3. Thread blocking is caused by *ObjectManagerThread*

Going back to *Figure 15*, which shows the stack trace for the blocked thread, and clicking on the link to *ObjectManagerThread* brings up the stack trace for the blocking thread. *Figure 18* shows that the code that is blocking the *DatabaseConnectorThread* is line number 26 of *ObjectManager.java*.

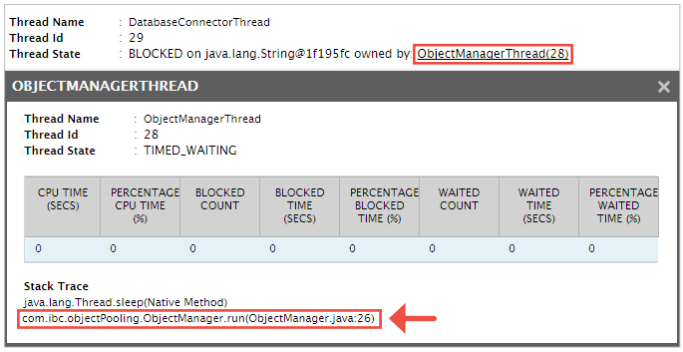

Figure 18: Stack trace for the blocked thread – ObjectManagerThread

Again, the use of text editor shows that *ObjectManagerThread* enters a 3600 second timed wait at line 26 (see *Figure 19*). This sleep call is inside a synchronized block with the local variable *mysync* being used as the object to synchronize on.

```
17   public void run()
18   {
19          synchronized(mysync)
20          {
21             long 1 = 0L;
22             while (!last)
23             {
24                 try
25                 {
26                     Thread.sleep(3600);
27                 }
28                 catch(Exception ex);
29                 {
30                     ex.printStackTrace();
31                 }
32              }
33           }
34       }
```

Figure 19: ObjectManager.java source file

```
123  public void getConnection()
124  {
125         synchronized(sync)
126         {
127            long 1 = 0L;
128            while (!finish1)
129            {
130                try
131                {
132                    Thread.sleep(3600);
133                }
134                catch(Exception ex);
135                {
136                    ex.printStackTrace();
137                }
```

Figure 20: DbConnection.java source file

Looking at the variable declarations at the top of each source code file (see *Figures 21 & 22*), one will quickly observe that both the *mysync* variable of the *ObjectManager* class and the sync variable of the *DbConnection* class in fact refer to the same static string, *test*.

```
1   package com.ibc.connectionPooling;
2
3   import com.ibc.objectPooling.*;
4   import java.util.Date;
5
6   public class DbConnection
7   {
8      public static boolean finish1 = false;
9
10     public static String sync = "test";
```

Figure 21: ObjectManager.java variable declarations

```
1   package com.ibc.objectPooling;
2
3   import com.ibc.objectPooling.*;
4   import java.util.Date;
5
6   public class ObjectManager extends Thread
7   {
8      public static boolean last = false;
9      public static String mysync = "test";
10     public ObjectManager()
```

Figure 22: DbConnection.java variable declarations

So, even though the programmer has given two different variable names in the two classes, the two classes refer to and are synchronizing on the same static string object, test. This is why two unrelated threads are interfering with each other's execution.

Troubleshooting

Modifying the two classes – ObjectManager and DbConnection – so that the variables mysync and sync point to two different strings by using the new String ("test") function resolves this problem.

End-to-End Application and Infrastructure Monitoring

Besides providing in-depth monitoring of the JVM, eG Enterprise offers in-depth monitoring of other tiers of the infrastructure as well. With monitoring support for over 180 enterprise applications, 10+ operating systems and 10+ hypervisors, eG Enterprise offers broad monitoring coverage for most enterprise and service provider infrastructures.

The monitoring can be done in an agent-based or an agentless manner. Administrators can choose which approach works for them. A single eG monitor (agent or agentless) can monitor all the JVMs running on a system, such the JVM used by web application servers like WebLogic, WebSphere, Tomcat, JBoss, SAP NetWeaver, etc. Moreover, eG Enterprise can also be used to monitor client/server Java applications and any custom applications using Java. Both Windows and UNIX-based environments (AIX, Linux, HP-UX, Solaris, etc.) are supported.

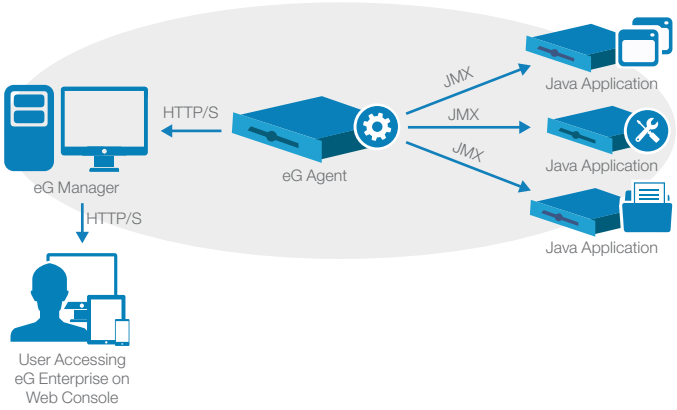

Figure 23: The eG Enterprise architecture

Additionally, eG enterprise offers code-level insight for Java applications by using byte code instrumentation and tracing how user transactions flow throughout the applications architecture. Administrators can easily identify which tier of the Java applications architecture is experiencing slowdown (see *Figure 24*).

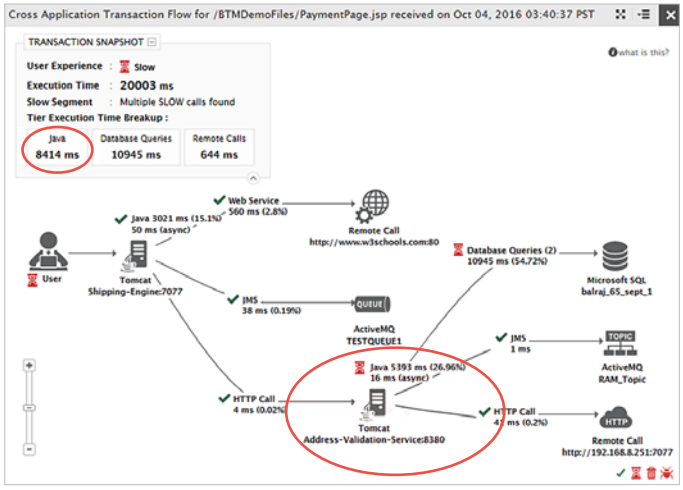

Figure 24: Java business transaction monitoring using eG Enterprise

In addition to real-time alerts and diagnosis, eG Enterprise also provides historical reports that are ideal for post-mortem diagnosis and pinpointing where the root-cause of a problem lies. Administrators can:

- Correlate Java application performance with that of the other infrastructure tiers and determine where the bottlenecks lie.
- Get insights into the performance of the JVM that can help them fine-tune the performance of the Java applications to ensure improved user experience.

Next Steps

For more information, please visit http://www.eginnovations.com/product/java-performance-monitoring, or email us at info@eginnovations.com

LIVE DEMO

Request a personal walkthrough to learn first hand how eG Enterprise can help improve performance and operations in your business environment.

FREE TRIAL

15-days of free monitoring and diagnosis, in your own infrastructure. Try it and learn exactly how eG Enterprise helps you ensure a great end-user experience and improve IT operations.

About eG Innovations

eG Innovations provides the world's leading enterprise-class performance management solution that enables organizations to reliably deliver mission-critical business services across complex cloud, virtual, and physical IT environments. Where traditional monitoring tools often fail to provide insight into the performance drivers of business services and user experience, eG Innovations provides total performance visibility across every layer and every tier of the IT infrastructure that supports the business service chain. From desktops to applications, from servers to network and storage, eG Innovations helps companies proactively discover, instantly diagnose, and rapidly resolve even the most challenging performance and user experience issues.

eG Innovations' award-winning solutions are trusted by the world's most demanding companies to ensure end user productivity, deliver return on transformational IT investments, and keep business services up and running. Customers include JP Morgan Chase, Citigroup, Depository Trust and Clearing Corporation, CSC, Cathay Bank, AllScripts, Honeywell, Fidelity Investments, Samsung, Xerox, Marathon Oil, US Department of State, The Government of Canada, McKesson, Aviva, AXA, and many more.

To learn more visit www.eginnovations.com.

Restricted Rights

The information contained in this document is confidential and subject to change without notice. No part of this document may be reproduced or disclosed to others without the prior permission of eG Innovations, Inc. eG Innovations, Inc. makes no warranty of any kind with regard to the software and documentation, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

© Copyright eG Innovations, Inc. All rights reserved.
All trademarks, marked and not marked, are the property of their respective owners.
Specifications subject to change without notice.

Total Performance Visibility