



**ORACLE®**

**Java Application Diagnostic Expert**



# Agenda

1. Enterprise Manager
2. Challenges
3. Java Application Diagnostics Expert (JADE)
4. Feature-Benefit Summary
5. Features Overview



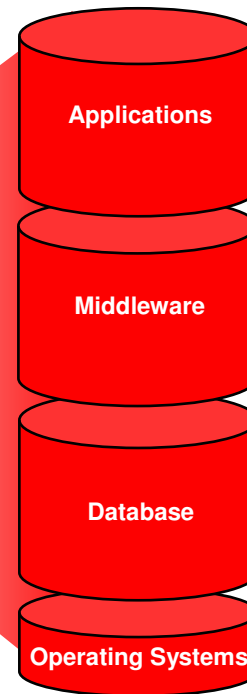
# Diagnostic Challenges

- Limited Production visibility
- Not enough depth to diagnose production issues
- Diagnostic tools not suited for production
- Inability to reproduce problems in other environments
- Hard to isolate culprit tier and component
- Impacts Time to Resolution

# Enterprise Manager - Java Application Diagnostic Expert (JADE)

## *Strengthening APM solution*

- Configuration Management
- Service Level Management
- **Application Performance Mgmt**
- Lifecycle Management
- Dashboards



### Java Application Diagnostic Expert (JADE)

- Diagnose problems in production
- View all java activity: instrumentation not needed
- Trace transactions across tiers, into the Oracle DB



# JADE Competitive Advantage

## BCI based tools

---

1. Complicated to install, deploy, use
  - Code instrumentation and server restarts
  - Experts needed to instrument and diagnose
2. High overhead (>20%) – Can't run in production
3. Silo oriented – No visibility from App Server through to DB

## ORACLE®

---

1. Very easy to install, deploy, and use
  - No modification of code or server restarts
  - Immediate visibility with no Application expertise required
2. Minimal Overhead (<1%) – Runs in production Servers
3. Transaction Visibility – Follow transactions from AS to Oracle database

# Feature-Benefit Summary

Key Features	Benefit
1. Hot deployment, without server restarts or configuration changes	<ul style="list-style-type: none"><li>• Quick time to value</li><li>• Simple to deploy</li><li>• Easy to use</li></ul>
2. Real-time visibility into JVM & Application state without instrumentation overhead	<ul style="list-style-type: none"><li>• Monitor complete Java stack including thread state and Java method/line visibility</li><li>• View of resource bottlenecks (DB, I/O, CPU, Locks, thread hangs etc)</li><li>• Low overhead allows production diagnosis</li><li>• No need to reproduce problems in test environment.</li><li>• No application knowledge required</li></ul>
3. Trace in-flight transactions	<ul style="list-style-type: none"><li>• View request performance problems in real time</li><li>• Analyze time spent in each resource bottleneck (extend to each java method)</li></ul>

## Feature-Benefit Summary - Continue

Key Features	Benefit
4. Cross-tier trace with Oracle DB	<ul style="list-style-type: none"><li>• Cross tier analysis of transactions between DB and middle-tier</li></ul>
5. Differential heap analysis	<ul style="list-style-type: none"><li>• Simplify memory growth and leak analysis</li></ul>
6. Request Performance	<ul style="list-style-type: none"><li>• Historical views to show the response times of end-user requests</li><li>• Response time breakdown by Servlet/JSP, EJB, JDBC and SQL</li></ul>
7. Interactive Transaction Trace	<ul style="list-style-type: none"><li>• Synthetic transaction response time profile</li></ul>
4. System monitoring	<ul style="list-style-type: none"><li>• Events and notification on all performance, load, and usage metrics</li><li>• Reports and historical trends</li></ul>

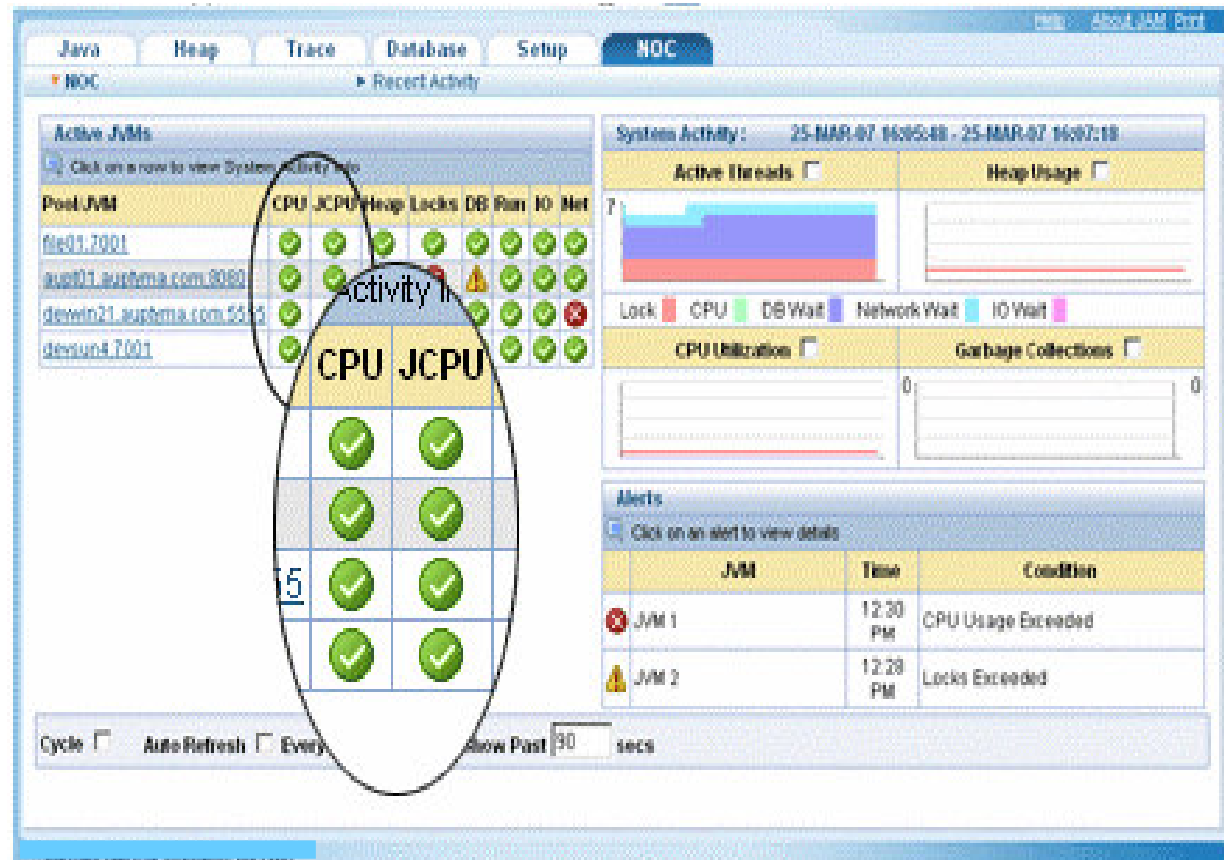
# 1. HOT Deployment

*Dynamic deployment on running applications*

1. No parameter changes needed
2. Deploys dynamically as a web application
3. Does not need app configuration changes
4. Does not need JVMPI/JVMTI
5. Does not mess with the class loader

## 2. JVM Monitoring

- Leading indicators of Performance
  - CPU, Heap Usage, GC Activity
  - Time spent in waiting for Object Locks, DB calls, IO, and Network



## 2. Time consuming Methods & Requests

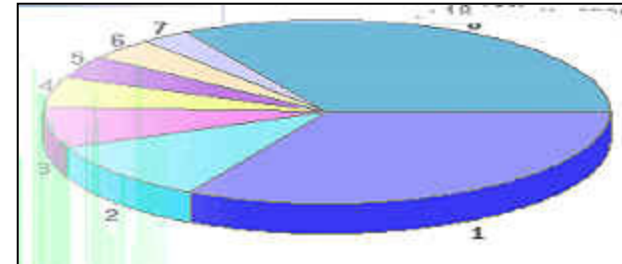
- Top methods including the complete call stack and cost for each method in the stack
- Top requests with associated cost

Top Requests	
Requests	Cost
<a href="#">/petstore/control/product</a>	1415794
<a href="#">/petstore/control/commitorder</a>	143
<a href="#">/petstore/control/prod</a>	140
<a href="#">/petstore/control/cart</a>	111
<a href="#">/petstore/control/category</a>	42
<a href="#">/petstore/control/productdetails</a>	42
<a href="#">/petstore/control/verifysignin</a>	39
<a href="#">/petstore/control/checkout</a>	25

	Method	Cost
	sun.reflect.NativeMethodAccessorImpl->invoke	95
	NetworkWar sun.reflect.NativeMethodAccessorImpl->invoke0	95
	org.apache.catalina.startup.Catalina->start	86
	org.apache.catalina.startup.Catalina->load	2
	org.apache.catalina.core.StandardContext->setPath	1
	org.apache.catalina.connector.Connector->setRedirectPort	4
	org.apache.catalina.connector.Connector->setPort	2
2.	com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser->endElement	30
3.	com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser->startElement	21
4.	sun.net.www.protocol.file.Handler->parseURL	15
6.	com.sun.jmx.mbeanserver.DynamicMetaDatumImpl->getMBeanInfo	12
6.	sun.reflect.GeneratedConstructorAccessor2->newInstance	10
7.	sun.misc.URLClassPath\$JarLoader->getResource	8

### 3. Transaction Response Time Analysis

- Trace active threads in real-time
- View the bottleneck resources



Stack Trace	View Totals By Percent
⊕ 0.35% java.lang.String->	View Totals By Percent
⊕ 0.35% java.lang.Object->clone	View Totals By Abs Numbers
⊖ 99.30% jamagent.jamrun->main	View Lock By Percent
⊖ 99.30% java.lang.reflect.Method->invoke	View Lock By Abs Numbers
⊖ 99.30% sun.reflect.DelegatingMethodAccessorImpl->invoke	View CPU By Percent
⊖ 99.30% sun.reflect.NativeMethodAccessorImpl->invoke	View CPU By Abs Numbers
⊖ 99.30% sun.reflect.NativeMethodAccessorImpl->invoke0	View DB By Percent
⊖ 99.30% org.apache.catalina.startup.Bootstrap->main	View DB By Abs Numbers
⊖ 81.12% org.apache.catalina.startup.Bootstrap->start	View IO By Percent
⊖ 81.12% java.lang.reflect.Method->invoke	View IO By Abs Numbers
⊖ 81.12% sun.reflect.DelegatingMethodAccessorImpl->invoke	View Net By Percent
⊖ 81.12% sun.reflect.NativeMethodAccessorImpl->invoke	
⊖ 81.12% sun.reflect.NativeMethodAccessorImpl->invoke0	
⊖ 81.12% org.apache.catalina.startup.Catalina->start	
⊖ 81.12% org.apache.catalina.core.StandardServer->start	
⊕ 72.38% org.apache.catalina.core.StandardService->start	
⊕ 8.74% org.apache.catalina.util.LifecycleSupport->fireLifecycleEvent	
⊕ 18.18% org.apache.catalina.startup.Bootstrap->load	

## 4. Cross-tier trace with Oracle DB

- Trace Java thread to DB session
  - Identify in-flight Java threads waiting for DB resource
  - Drill to SQL
- Trace DB session to Java thread
  - View DB sessions waiting for or holding locks
  - Identify Java thread holding DB session

The screenshot shows the Oracle JRM interface with several key components:

- JVM Activity Table:**

Machine	OS Name	JVM	Port	CPU	OSR	Mem	Run	DB	...
derwin21.aupharma.com	Windows 2003	Sun 1.5.0_05	12501	0	18	34	0	0	...
sup801	Linux	Sun 1.5.0_06	3104	55	2	5	0	2	...
- DB States Table:**

DBStates	Cost
enqueue	122
db file scattered read	30
- SQL Statement:**

```
SELECT userid , status , email , firstname , lastname , addr1 , addr2 , city , state , zip , country
, phone
FROM account
WHERE userid = :1
```

Callouts in the image:

- "Identify DB Wait" points to the DB column in the JVM Activity table.
- "DB State Causing the Bottleneck" points to the 'enqueue' row in the DB States table.
- "SQL Causing the problem" points to the SQL statement window.

# 5. Differential Heap Analysis

- Take Heap snapshots with minimal overhead
- Memory leak analysis by comparing two different heap snapshots taken over a period of time

The screenshot shows the Oracle JAM interface for differential heap analysis. The top window, 'Compare Heap Usage by Roots', compares two heap snapshots: 'demo!nc:7001 26-JAN-06 13:49 68/155 MB' (Heap 1) and 'demo!nc:7001 26-JAN-06 13:49 61/155 MB' (Heap 2). The table below shows the difference in object counts and sizes.

Root	Objects	KB	Objects	KB	Delta
System Classes	1213177	57,373	813135	46,543	10,830
ThreadLocals	165319	10,071	165319	10,071	0
ProtectionDomains	165186	10,019	165186	10,019	0
ThreadOther	165170	10,033	165170	10,033	0
ObjectMonitor	165069	10,015	165069	10,015	0
System Other	165065	10,014	165065	10,014	0

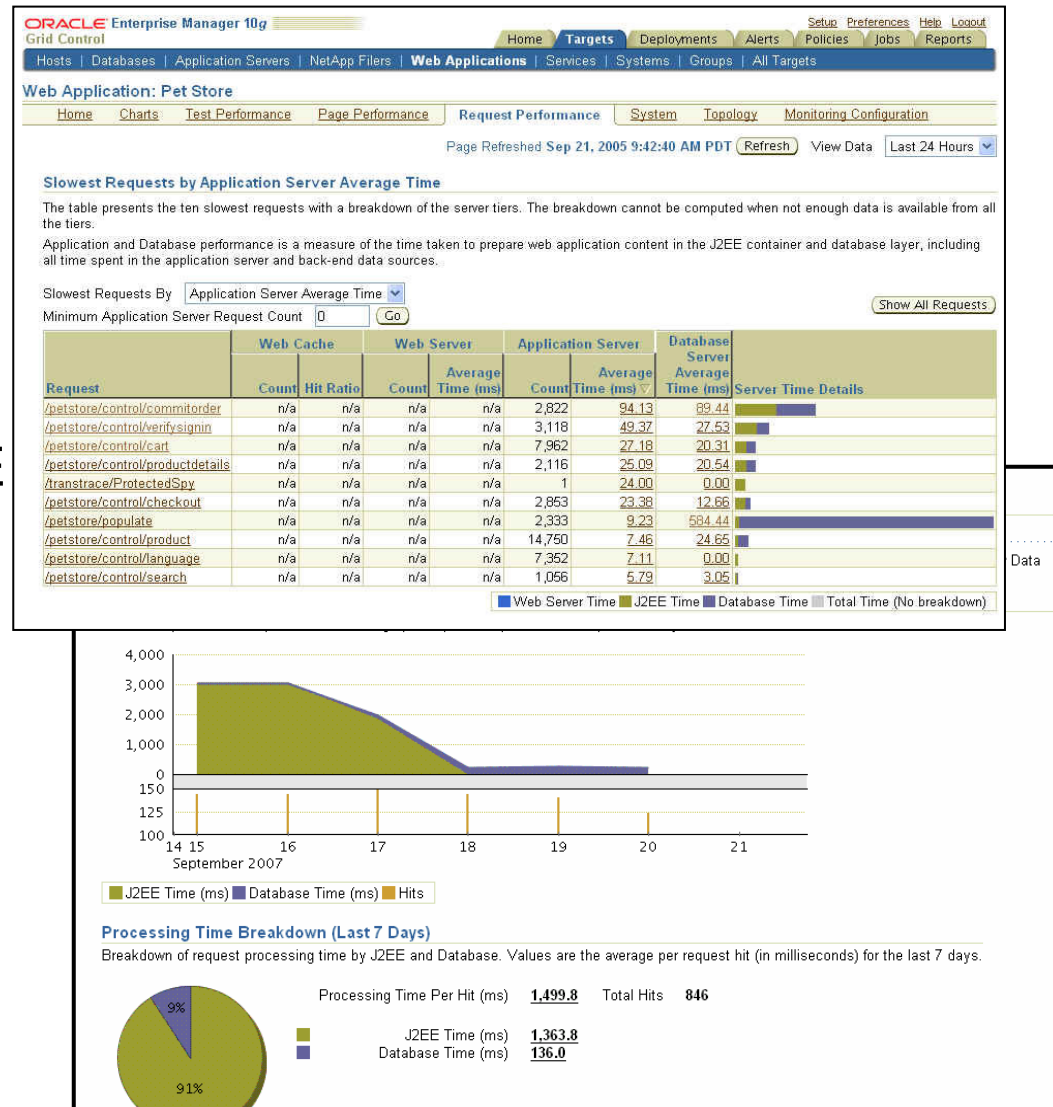
The bottom window, 'Top 40 objects in System Classes', provides a detailed breakdown of the 'System Classes' heap. It compares Heap 1 and Heap 2 across various object types and signatures, showing the change in count, bytes, children, and adjusted KB (Delta).

Type	Signature	Heap 1					Heap 2					Delta	
		Count	Bytes	Children	KB	Adj	Count	Bytes	Children	KB	Adj	KB	Adj
Class	java/lang/Object	1	240	1,213,175	57,373	12,768,283	1	240	813,133	46,543	10,650,604	10,830	2,117,679
Instance	java/lang/Class	1	64	165,065	10,014	4,089,895	1	64	165,065	10,014	4,089,863	0	32
Array	[]	1	72	15	2	804	1	72	15	2	804	0	0
Array	[S]	2	32	2	0	32	2	32	2	0	32	0	0
Array	[I]	1	16	1	0	16	1	16	1	0	16	0	0
Class	Null	2	344	2,171	227	89,614	2	344	2,171	227	89,614	0	0
Constant Pool	Null	1	328	2	0	416	1	328	2	0	416	0	0
Symbol	Null	2	64	2	0	64	2	64	2	0	64	0	0
Class	java/sql/Ref	1	256	16	1	547	1	256	16	1	547	0	0

A circular callout highlights the 'Delta KB' value of 10,830 for the 'java/lang/Object' class, indicating the memory increase between the two snapshots.

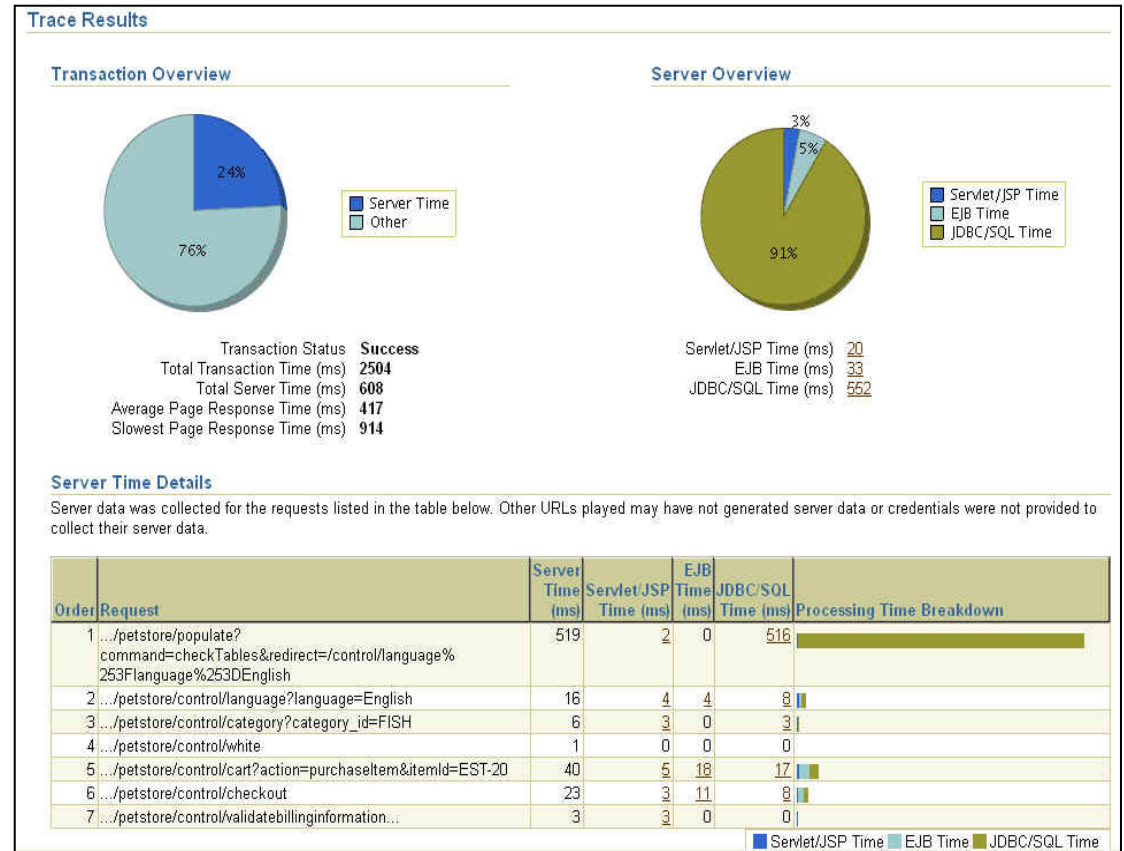
# 6. Request Performance → JVM Diagnostics

- For web applications (URL), capture time spent in each tier
- View performance historically. Identify if J2EE time is the cause
- Drill-down to JVM (JADE) to diagnose the root cause of increased J2EE time
- Identify problems to method, to resources, or due to memory leaks



# 7. Interactive Transaction Trace

- Web Application Transactions (URL) - Playback interactively
- View the time spent in server and network
- View response time breakdowns by Servlet/JSP, EJB, JDBC, and SQL
- Drilldown to JVM (JADE) to diagnose problems in JVM



# 8. System Monitoring

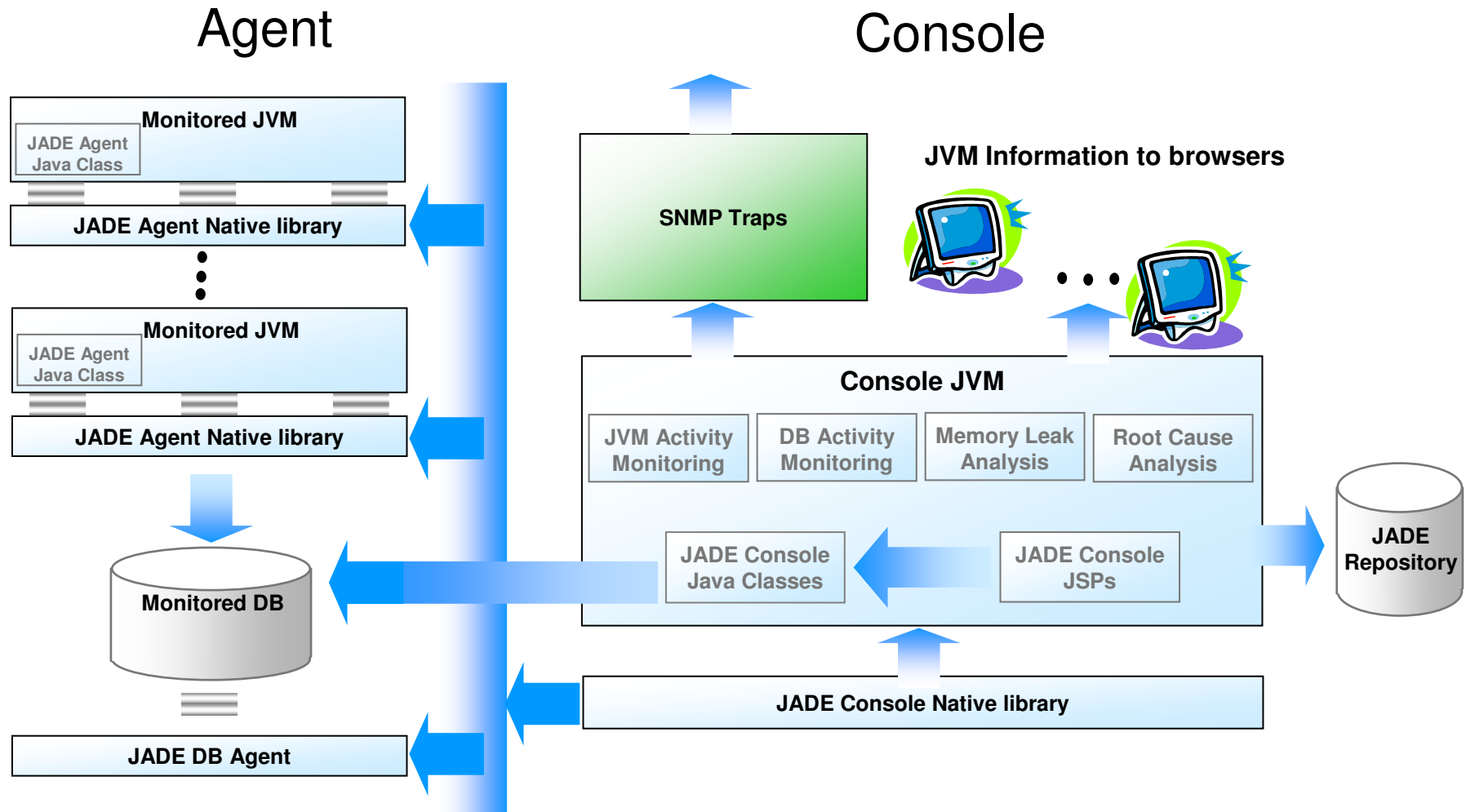
- 24x7 monitoring of availability, performance, load, and usage metrics of Application Server and the host
- Blackouts, events, notifications, and metrics comparison
- Historical trends of metrics and events
- Reports



# In Summary: Typical problems addressed

1. Application running slow because of serialized DB connections causing long wait times for Java locks
2. Mid-tier CPU utilization is high → identify top classes/methods
3. When a user request is hung, JADE allows you to identify the affected current state and line of code in the application
4. When GC is running often, consuming resources & slowing application, JADE can identify the source of the memory leak
5. Identify java thread associated with long running SQL
6. Request performance scenario

# JADE Architecture



# Platform Support for JADE

## Supported JVMs

- Solaris 8-10 on SPARC
  - Sun 1.3.1, 1.4.2, 1.5.0, 1.6.0
- Solaris 9-10 on X86
  - Sun 1.5.0
- Linux on x86
  - Sun 1.3.1, 1.4.2, 1.5.0, 1.6.0
  - IBM 1.3.1, 1.4.2
- AIX 5i on PowerPC
  - IBM 1.3.1, 1.4.2
- HPUX 11i on PARISC
  - HP 1.3.1, 1.4.2
- Windows 2K, XP, 2003 on x86
  - Sun 1.4.2, 1.5.0

## Application Servers

- Oracle Application Server
- BEA WebLogic
- IBM WebSphere
- SAP NetWeaver
- Tomcat
- Apache
- Resin
- JBoss

## Oracle 8i-10g for cross-tier correlation

- Linux, HP-UX, Solaris, AIX



# Java Application Diagnostic Expert (JADE)



**Shashi Senger**



# Problems

- Mission Critical business applications were experiencing daily down time for 3+ weeks/quarter
- **Reduced Availability** identified by monitoring tools (EMAN and Gomez)
- Java Problems:
  - JVM was either in hung state or was responding very slow
  - User transactions were hanging or running slow and users had to **restart all the work** all over again
  - Database locking issues bringing down databases impacting other common resources and applications
- Availability was always **below 97%** for 4+ weeks/quarter



# Applications & Problems

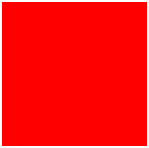
- OT Production
  - OT/B2B (OT, ICT and Bondi WebServices) - Memory leak in SCclient
- CCX-LT
  - X1 - High memory usage
  - CSCC (Quote and OCM) - High CPU Usage
  - QB/SX - High memory Usage
  - Status Tool - High memory Usage
  - Selectica SC - Locking issues bringing CMRS and other tools down
  - Selectica SC - CPU burn and memory usage

**Reproducing these issues in Test/Dev environment was extremely painful and time consuming**



# JADE Deployment

- JADE was deployed on Selectica by changing the startup script
- On IBM WAS it was deployed as a WAR file
- No instrumentation was needed, immediate visibility after deployment
- No performance overhead due to JADE
- JADE provided rapid visibility into critical issues
  - Heap Growth resulting in Out of Memory
  - Hanging Transactions



# Problem 1 – Memory Leak (OutOfMemory)



# Memory Leak (OutOfMemory)

- Monitoring solutions showed
  - Initially when heap size went to 75%, full GC would bring the memory down
  - However after some time, full GC would not be able to free any memory and full GCs would run continuously, bringing the system to a halt
- BCI based diagnostic tool would crash the JVM when trying to diagnose memory leak



# Memory Analysis with JADE

- Able to see a hash table growing inside the JVM
- When the hash table was small, as the garbage accumulated, full GC would successfully free memory when heap size approached 75%
- As the hash table itself grew to 75% of the heap, the full GC could not free any memory and became ineffective
- This explained why we were seeing full GCs free memory initially and not be able to free memory later



# Tuning JVM memory

- Identifying the reachable path of the hash table helped us identify the cache within Selectica which was growing
- Once the cache was identified, Selectica was able to provide us with settings for controlling the cache
- Using JADE we were able to fine tune the different cache settings for optimal performance



# Problem 2 – Application Hangs



# Application Hangs

- Response times degrade substantially
- Configurator tool was a standalone Java application
- Unable to determine if the slowdown was in Java or the database
- Spent lots of effort and consulting dollars trying to identify the bottleneck with BCI based tool

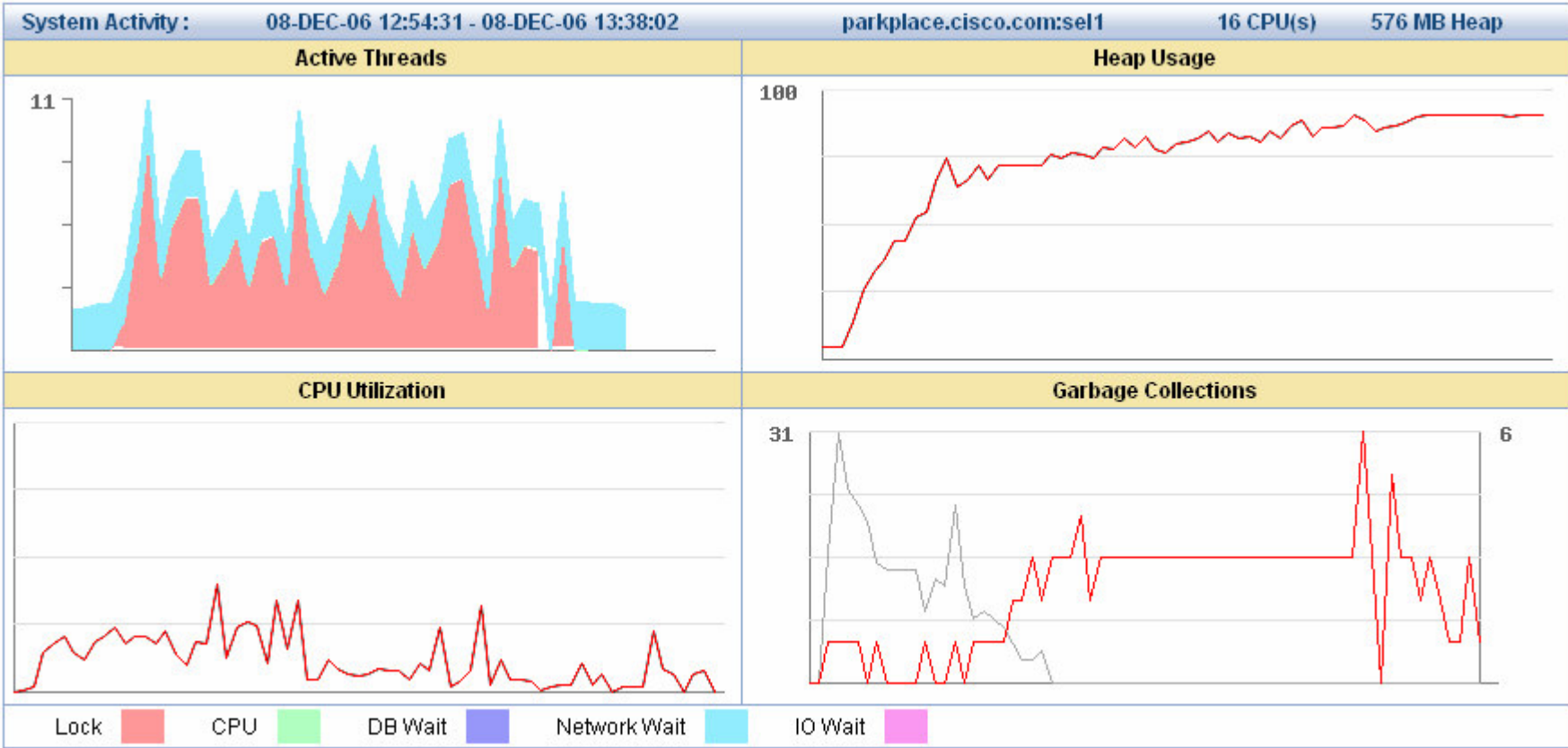


# Diagnose Hanging Issue with JADE

- After deployment we noticed a lot of thread contention from JADE
- JADE identified a custom callout responsible for locking
- We were able to pinpoint the call in less than a day (as soon as the issue occurred)



# Before





# After

