LIFERAY.

Performance Monitoring and Tuning

- **Liferay Chicago User Group (LCHIUG)**
- **James Lefeu**
- **29AUG2013**

Outline

I. Definitions

II. Architecture

III.Requirements and Design

IV.JDK Tuning

V. Liferay Tuning

VI.Profiling and Monitoring Tools

VII.Setting up a Performance Test

I. Definitions

I. Definitions

- Optimizing:
  - Make the best or most effective use of (a situation, opportunity, or resource).
  - Rearrange or rewrite (data) to improve efficiency of retrieval or processing.
    - http://www.google.com/#q=optimizing
  - To make as perfect or effective as possible.
  - To increase the computing speed and efficiency of (a program), as by rewriting instructions.
    - http://www.thefreedictionary.com/optimizing

I. Definitions

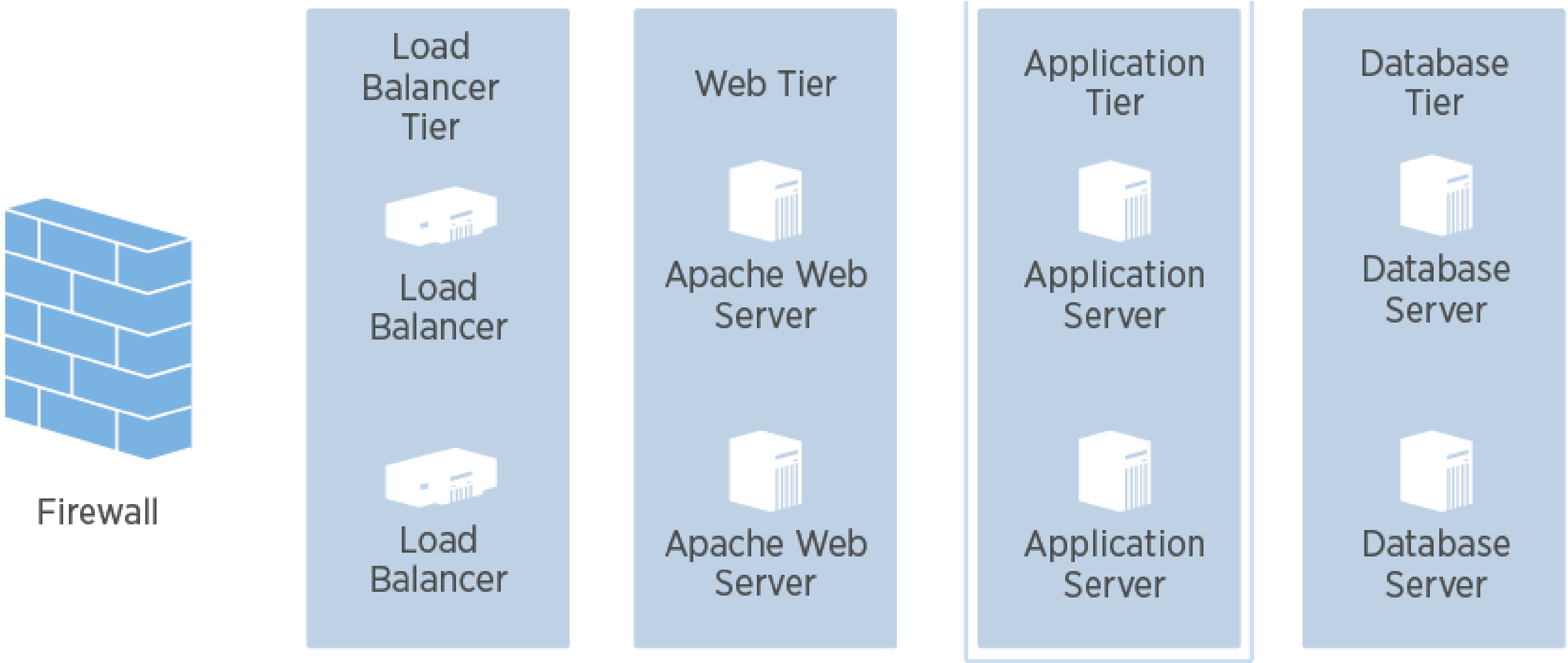
- Performance Tuning
  - Systematic tuning follows these steps:

1. Assess the problem and establish numeric values that categorize acceptable behavior.
2. Measure the performance of the system before modification.
3. Identify the part of the system that is critical for improving the performance. This is called the bottleneck.
4. Modify that part of the system to remove the bottleneck.
5. Measure the performance of the system after modification.
6. If the modification makes the performance better, adopt it. If the modification makes the performance worse, put it back the way it was

II. Architecture

II. Architecture

II. Architecture



LPEE Reference Architecture

II. Architecture

- Do I have enough CPU power?
- Do I have enough RAM?
- Do I have enough disk space?
- Is the network fast enough?
- Is the OS the best one to use?
- Is the OS configured optimally?
- Are the applications I'm using the best ones to use?
- Are the applications I'm using configured optimally?

III. Requirements and Design

III. Requirements and Design

- What is absolutely minimally necessary?
- What are you trying to maximize?
- What are you trying to minimize?
- What is an appropriate balance between differing optimization factors?
  - RAM vs. CPU usage
  - Security vs. Efficiency

III. Requirements and Design

- Response Time Requirements
  - How long is an acceptable delay for us to process a single request?
  - Do different types of requests have different response requirements?
  - Is it acceptable to simply say “Please wait while the server responds”?
  - Are “pauses” of any amount acceptable?

III. Requirements and Design

- Throughput Requirements
  - How many requests can we maximally handle at the same time (each request simply logging in)?
  - What minimum number of requests must we be able to handle concurrently (each running the most performance intensive features of Liferay)?
  - Is there a specified number of Transactions Per Second (TPS) which are needed either at the application level or at the system level?

III. Requirements and Design

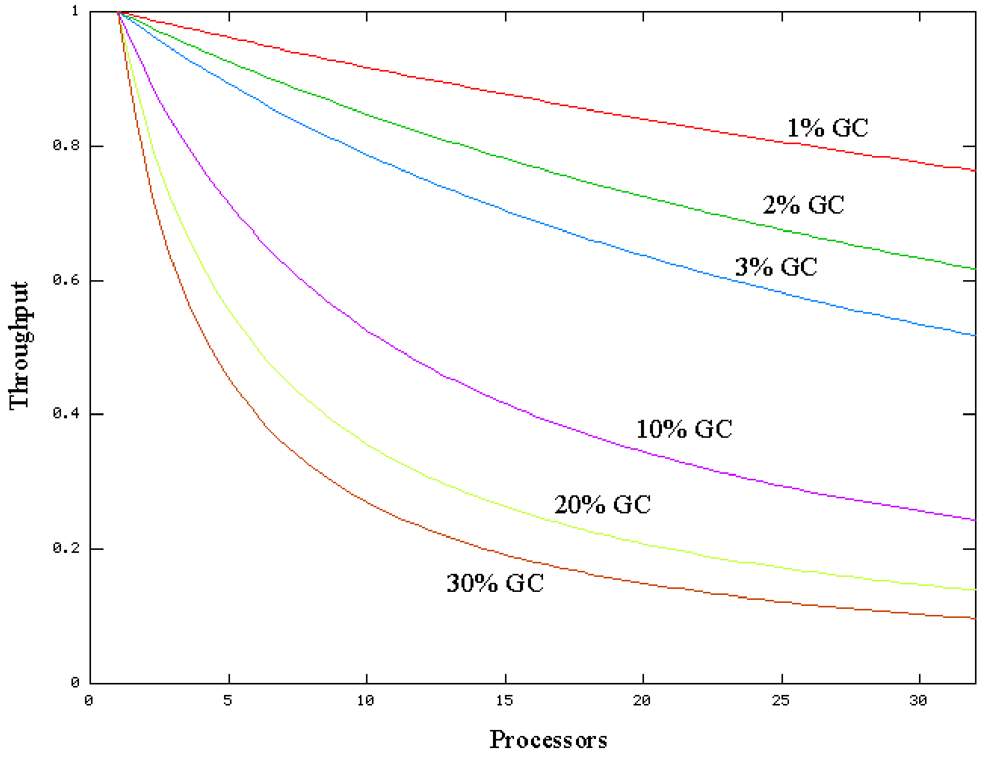
- Load Requirements
  - How much CPU, Memory, and I/O should be consumed in the best, average, and worst cases?
  - How should the system respond when we become near, reach, or go beyond the acceptable limits?

III. Requirements and Design

- Uptime Requirements
  - Is redundancy needed to maintain uptime?
  - Is it acceptable for a server to go down for a short period of time?
  - How much service delay is acceptable when in a transition from a failed server to a backup server?
  - Which types of outages do we plan to prevent against?

IV. JDK Tuning

IV. JDK Tuning



IV. JDK Tuning

- Reasons to tune the JDK
  - “Out of memory” or “concurrent mode failure”
    - Solved by increasing GC size
  - Throughput is drastically affected by large % of time in GC.
    - Solved by decreasing GC size

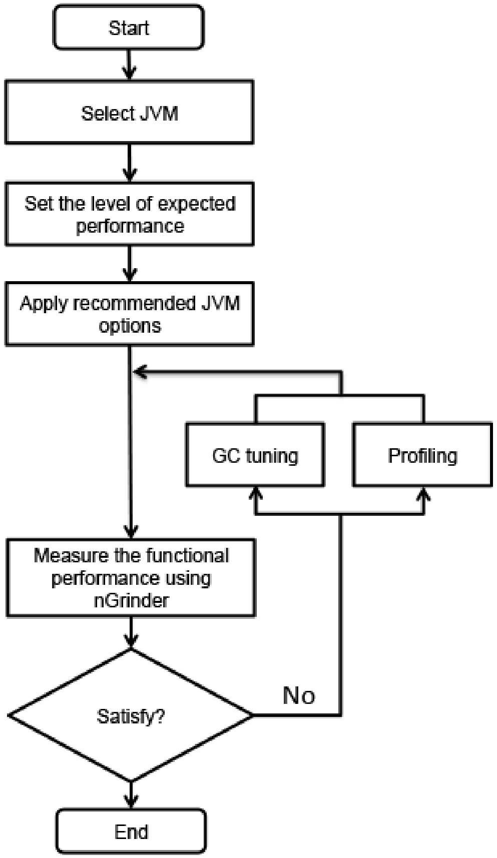
IV. JDK Tuning

- You want to have enough GC heap space to handle your high capacity situations.
- But, you don't want to have so much GC heap such that your throughput does down.
- One option is to try to split the application across multiple JVMs to keep smaller heap spaces.
- Many problems are solved **not** by GC tuning, but through application code refactoring. Thus, GC tuning should be done when application code refactoring is not an option.

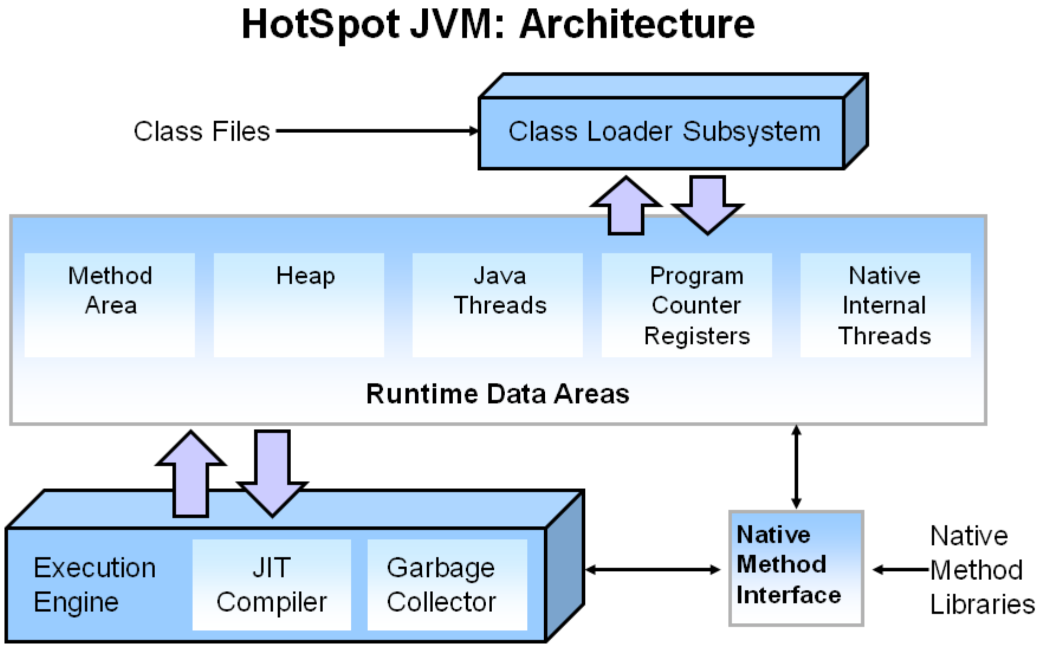
IV. JDK Tuning

- If the logs show just 0.1-0.3 seconds in GC, then you don't need to tune the GC.
- If the logs show 1-3 seconds or more, GC tuning is necessary.
- If the logs show less than 1% of the time spent in GC, then you don't need to tune the GC.
- If the logs show more than 10% of the time spent in GC, then GC tuning is necessary.

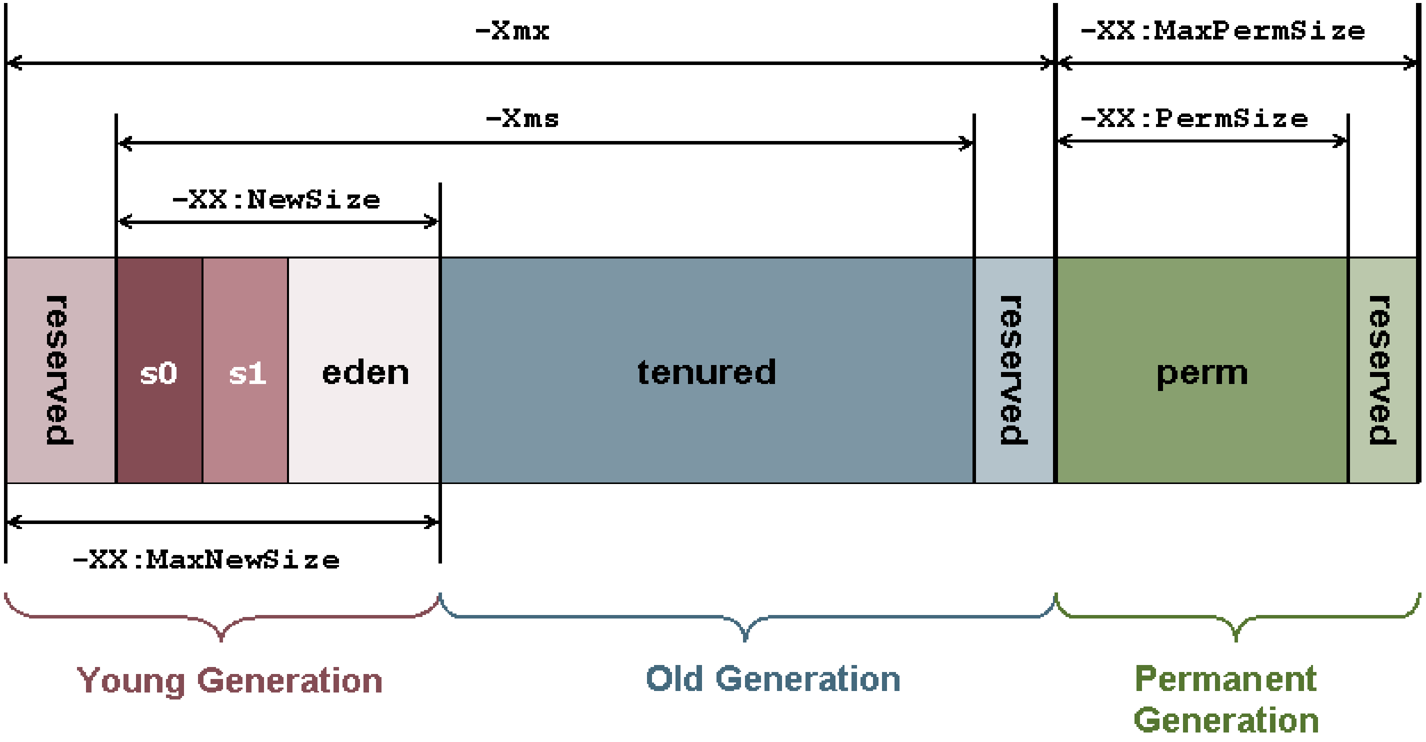
IV. JDK Tuning


Start
Select JVM
Set the level of expected performance
Apply recommended JVM options
GC tuning
Profiling
Measure the functional performance using nGrinder
Satisfy?
No
End

IV. JDK Tuning



IV. JDK Tuning



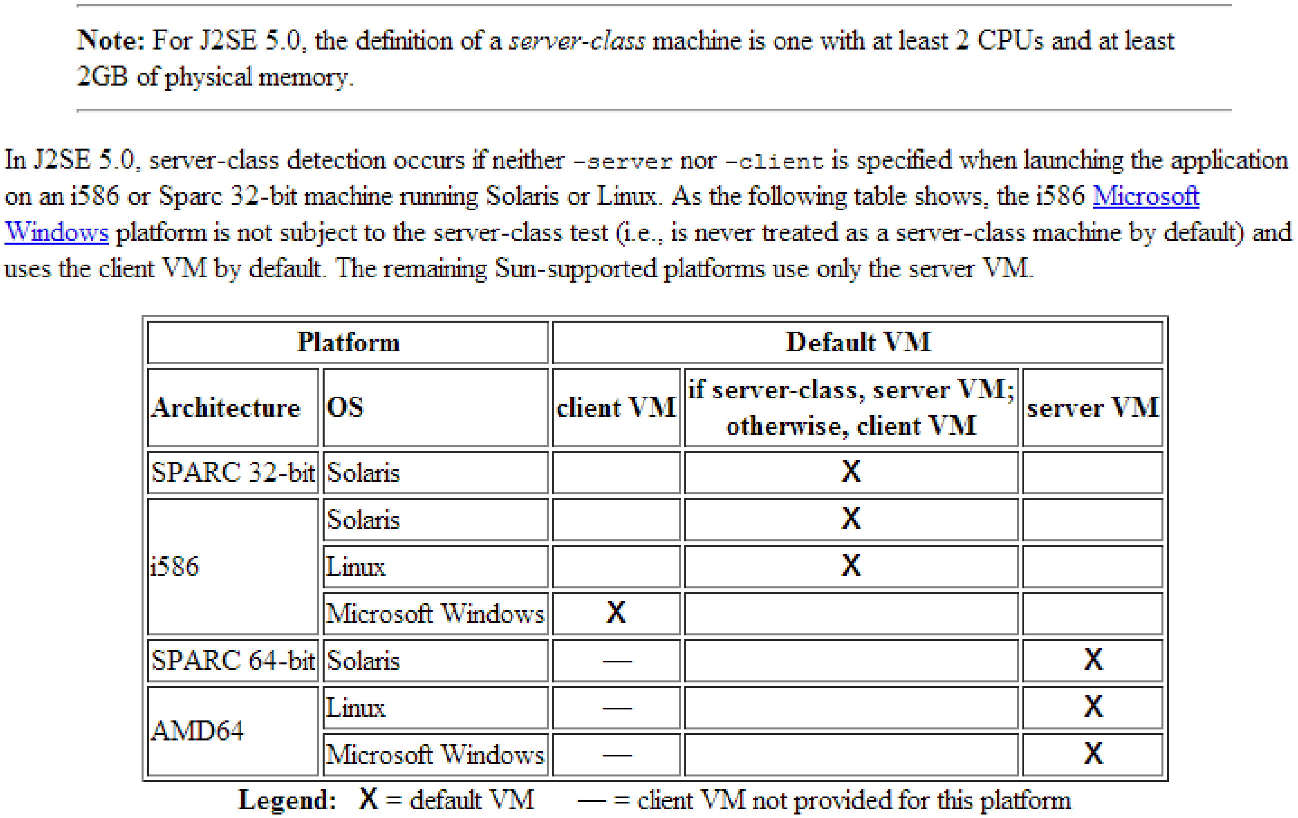
IV. JDK Tuning

- -verbose:gc
  - [GC 325407K->83000K(776768K), 0.2300771 secs]
  - [GC 325816K->83372K(776768K), 0.2454258 secs]
  - [Full GC 267628K->83769K(776768K), 1.8479984 secs]
- -XX:+PrintGCDetails
  - [GC [DefNew: 64575K->959K(64576K), 0.0457646 secs] 196016K->133633K(261184K), 0.0459067 secs]
- -XX:+PrintGCTimeStamps
  - 111.042: [GC 111.042: [DefNew: 8128K->8128K(8128K), 0.0000505 secs]111.042: [Tenured: 18154K->2311K(24576K), 0.1290354 secs] 26282K->2311K(32704K), 0.1293306 secs]

IV. JDK Tuning

- Client JVM
  - The Java HotSpot Client VM has been specially tuned to reduce application start-up time and memory footprint, making it particularly well suited for client environments.
- Server JVM
  - The Server VM contains an advanced adaptive compiler that supports many of the same types of optimizations performed by optimizing C++ compilers, as well as some optimizations that cannot be done by traditional compilers, such as aggressive inlining across virtual method invocations.

IV. JDK Tuning



Note: For J2SE 5.0, the definition of a *server-class* machine is one with at least 2 CPUs and at least 2GB of physical memory.

In J2SE 5.0, server-class detection occurs if neither `-server` nor `-client` is specified when launching the application on an i586 or Sparc 32-bit machine running Solaris or Linux. As the following table shows, the i586 Microsoft Windows platform is not subject to the server-class test (i.e., is never treated as a server-class machine by default) and uses the client VM by default. The remaining Sun-supported platforms use only the server VM.

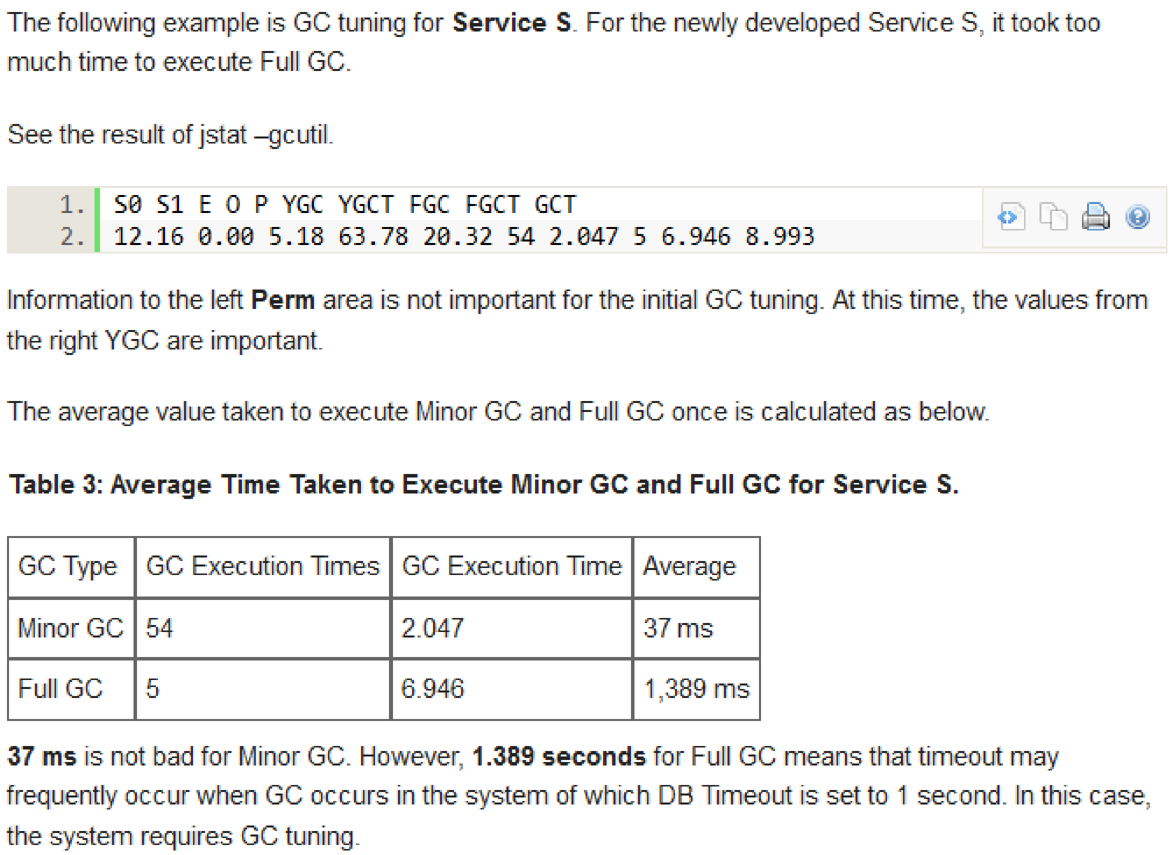
Platform		**Default VM**		
Architecture	**OS**	**client VM**	**if server-class, server VM; otherwise, client VM**	**server VM**
SPARC 32-bit	Solaris		X	
i586	Solaris		X	
	Linux		X	
	Microsoft Windows	X		
SPARC 64-bit	Solaris	—		X
AMD64	Linux	—		X
	Microsoft Windows	—		X

Legend: X = default VM — = client VM not provided for this platform

IV. JDK Tuning

- If your OS supports it, enabling “Huge Page” can significantly improve the JVM's GC and memory accessing performance.
  - If you run the command “less /proc/meminfo” and see a “HugePageSize” entry, then your OS does support it.
  - If so, edit “/etc/sysctl.conf” and add “vm.nr_hugepages = 1024”
  - Then, run “sysctl -p” to make the changes effective.
  - Rerunning “less /proc/meminfo” will show the “HugePages_Total” and “HugePages_Free” entries having a value greater than zero.

IV. JDK Tuning - Example



The following example is GC tuning for **Service S**. For the newly developed Service S, it took too much time to execute Full GC.

See the result of jstat –gcutil.

```
S0 S1 E O P YGC YGCT FGC FGCT GCT
12.16 0.00 5.18 63.78 20.32 54 2.047 5 6.946 8.993
```

Information to the left **Perm** area is not important for the initial GC tuning. At this time, the values from the right YGC are important.

The average value taken to execute Minor GC and Full GC once is calculated as below.

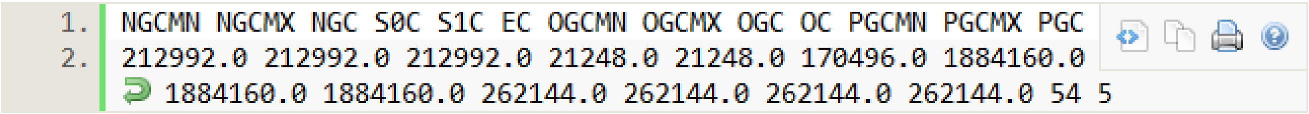
Table 3: Average Time Taken to Execute Minor GC and Full GC for Service S.

GC Type	GC Execution Times	GC Execution Time	Average
Minor GC	54	2.047	37 ms
Full GC	5	6.946	1,389 ms

37 ms is not bad for Minor GC. However, **1.389 seconds** for Full GC means that timeout may frequently occur when GC occurs in the system of which DB Timeout is set to 1 second. In this case, the system requires GC tuning.

IV. JDK Tuning - Example

First, you should check how the memory is used before starting GC tuning. Use the jstat –gccapacity option to check the memory usage. The result checked from this server is as follows.



```
NGCMN NGCMX NGC S0C S1C EC OGCMN OGCMX OGC OC PGCMN PGCMX PGC
212992.0 212992.0 212992.0 21248.0 21248.0 170496.0 1884160.0
   1884160.0 1884160.0 262144.0 262144.0 262144.0 262144.0 54 5
```

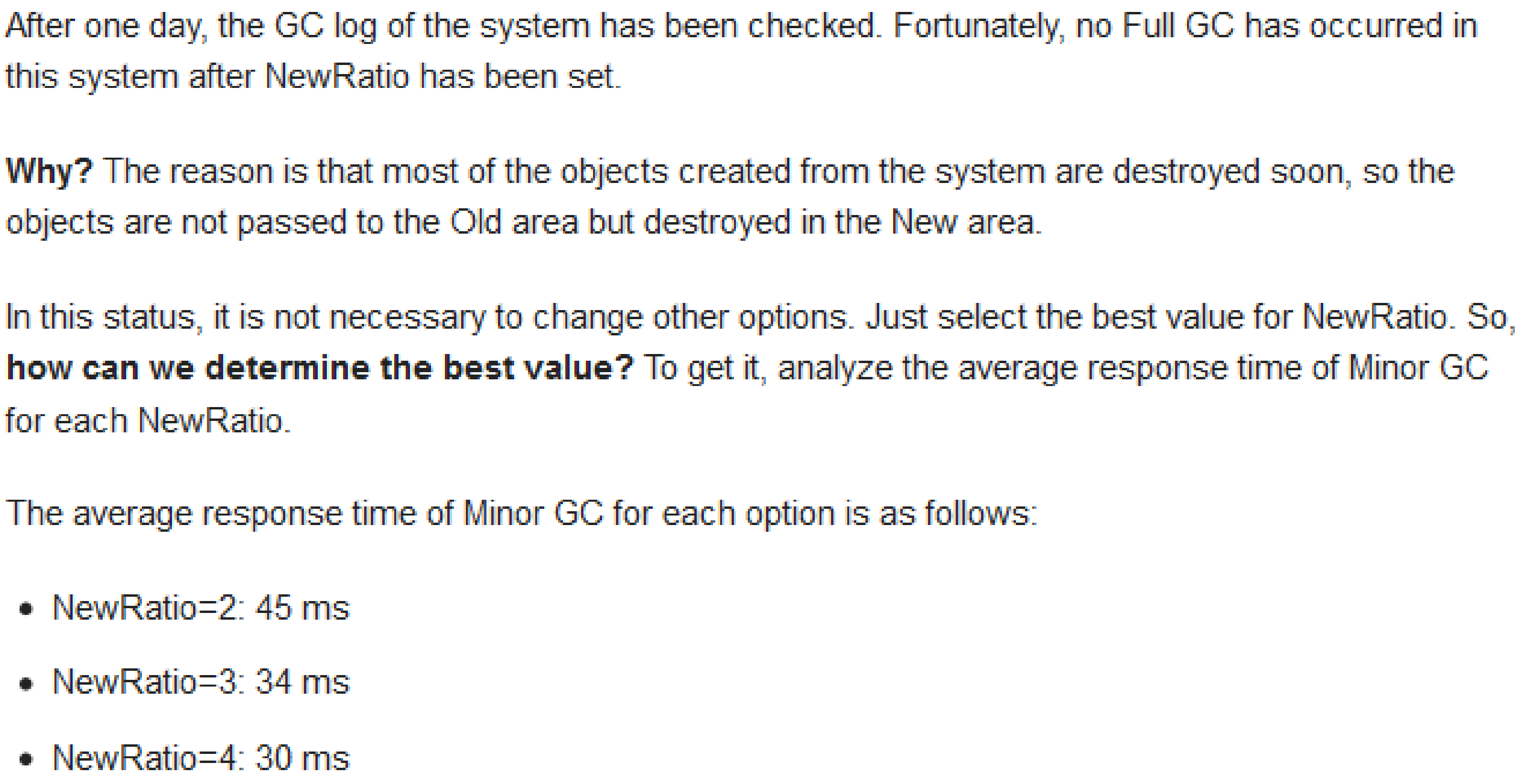
The key values are as follows.

- New area usage size: 212,992 KB
- Old area usage size: 1,884,160 KB

Therefore, the totally allocated memory size is 2 GB, excluding the Perm area, and New area:Old area is 1:9. To check the status in a more detailed way than **jstat**, the -verbosegc log has been added and three options were set for the three instances as shown below. No other option has been added.

- NewRatio=2
- NewRatio=3
- NewRatio=4

IV. JDK Tuning - Example



After one day, the GC log of the system has been checked. Fortunately, no Full GC has occurred in this system after NewRatio has been set.

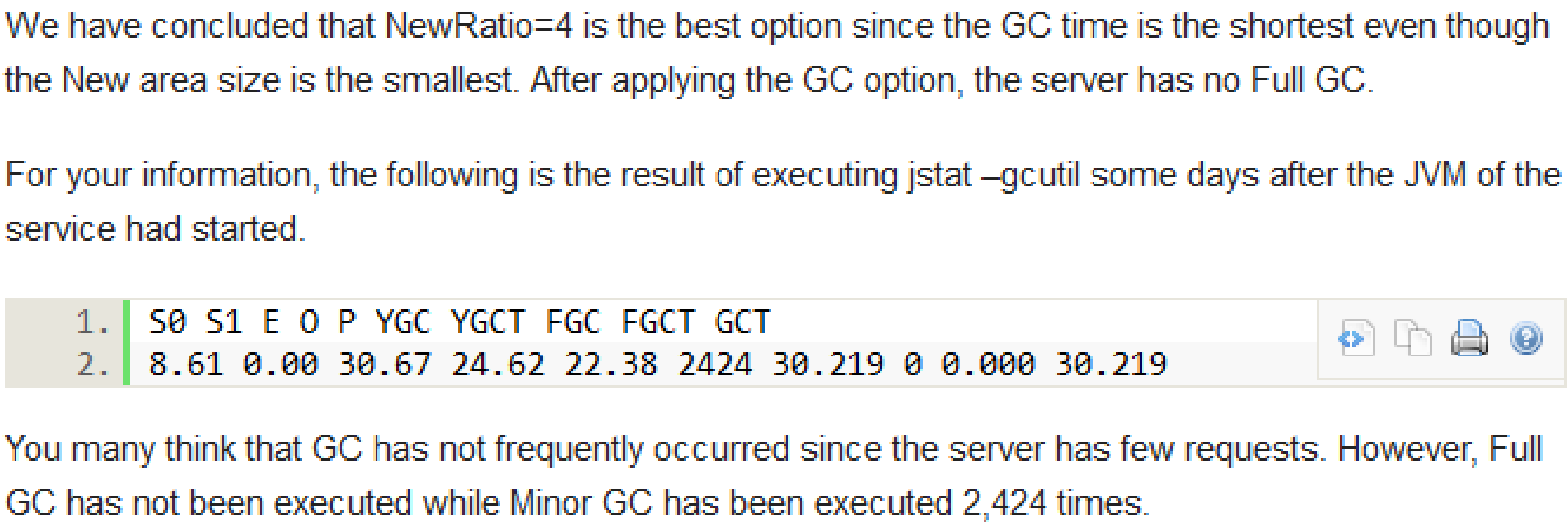
Why? The reason is that most of the objects created from the system are destroyed soon, so the objects are not passed to the Old area but destroyed in the New area.

In this status, it is not necessary to change other options. Just select the best value for NewRatio. So, **how can we determine the best value?** To get it, analyze the average response time of Minor GC for each NewRatio.

The average response time of Minor GC for each option is as follows:

- NewRatio=2: 45 ms
- NewRatio=3: 34 ms
- NewRatio=4: 30 ms

IV. JDK Tuning - Example



We have concluded that NewRatio=4 is the best option since the GC time is the shortest even though the New area size is the smallest. After applying the GC option, the server has no Full GC.

For your information, the following is the result of executing jstat –gcutil some days after the JVM of the service had started.

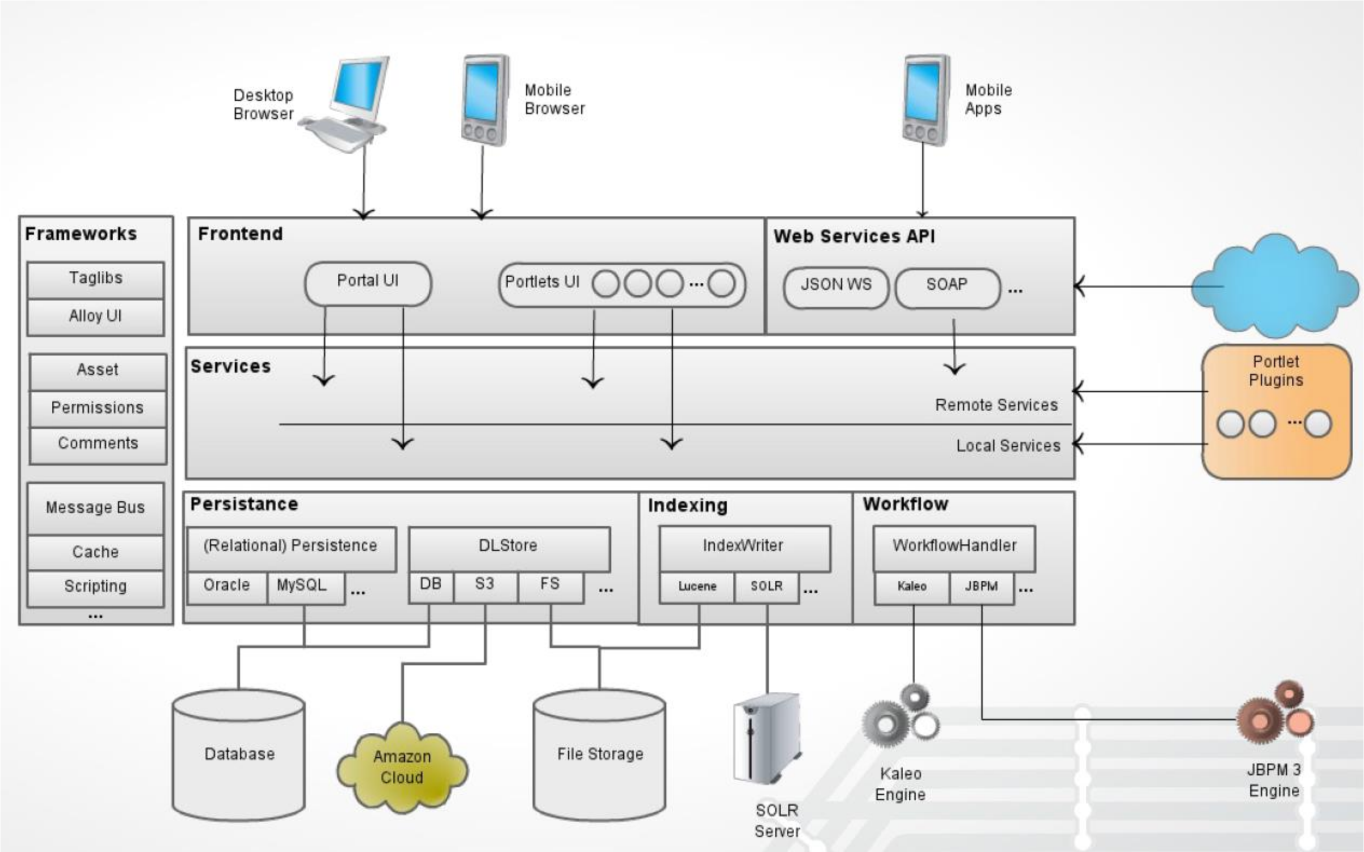
```
S0 S1 E O P YGC YGCT FGC FGCT GCT
8.61 0.00 30.67 24.62 22.38 2424 30.219 0 0.000 30.219
```

You many think that GC has not frequently occurred since the server has few requests. However, Full GC has not been executed while Minor GC has been executed 2,424 times.

http://architects.dzone.com/articles/how-tune-java-garbage

V. Liferay Tuning

V. Liferay Tuning - Architecture



V. Liferay Tuning - Architecture

L1 – “chip level cache”

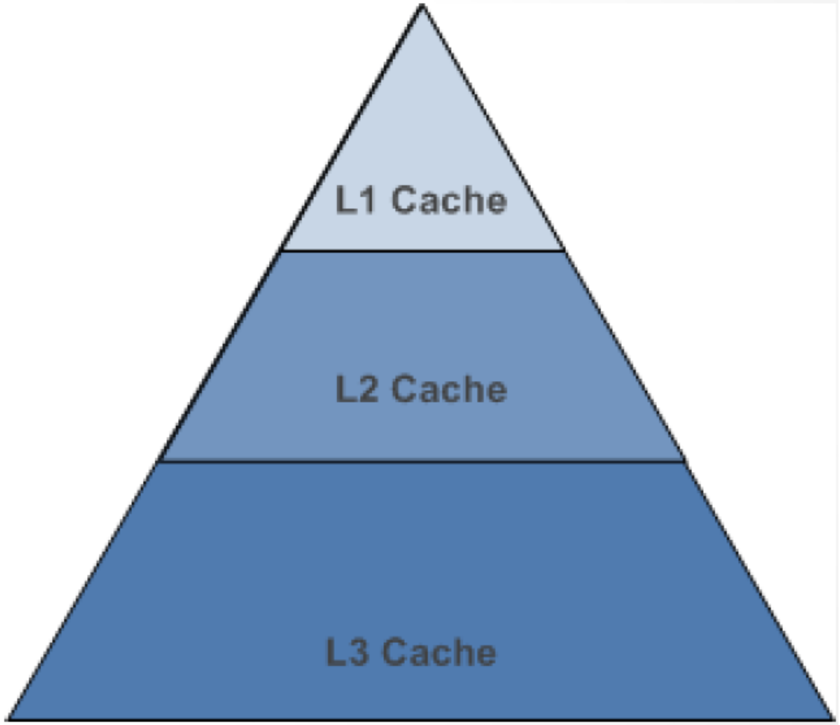
Request scoped cache

L2 - “system memory”

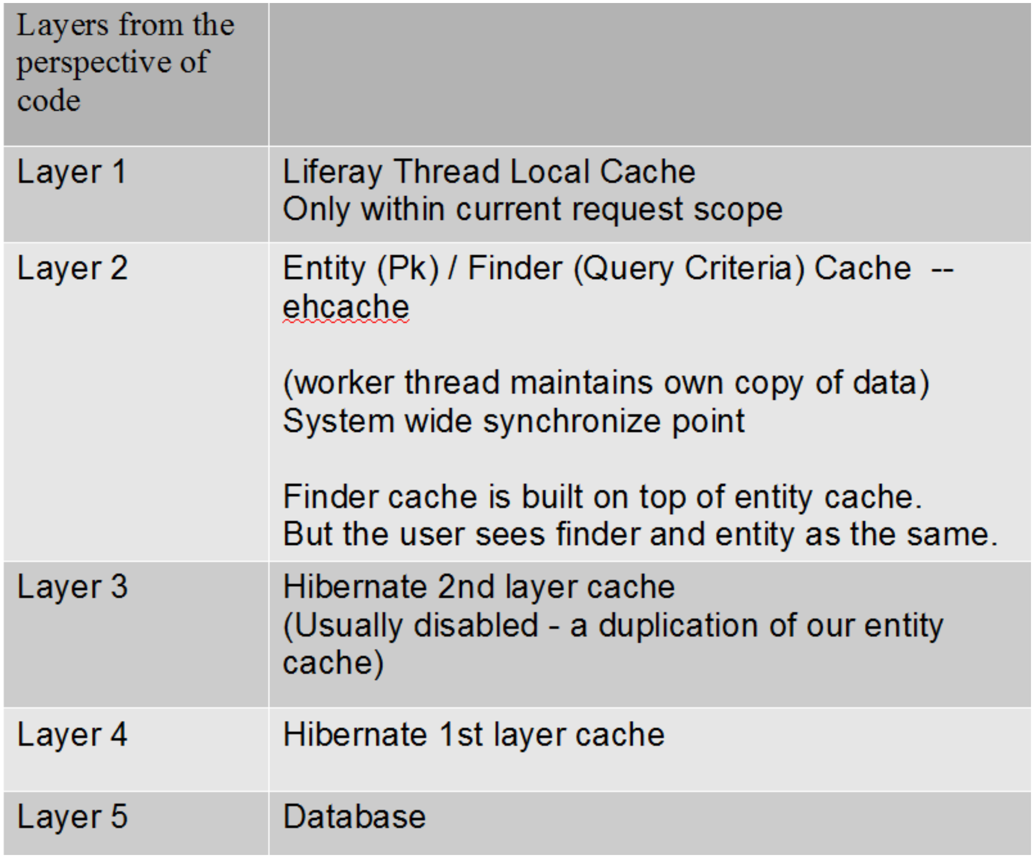
Constrained by heap size

L3 – “swap space”

Equivalent to virtual memory swap space

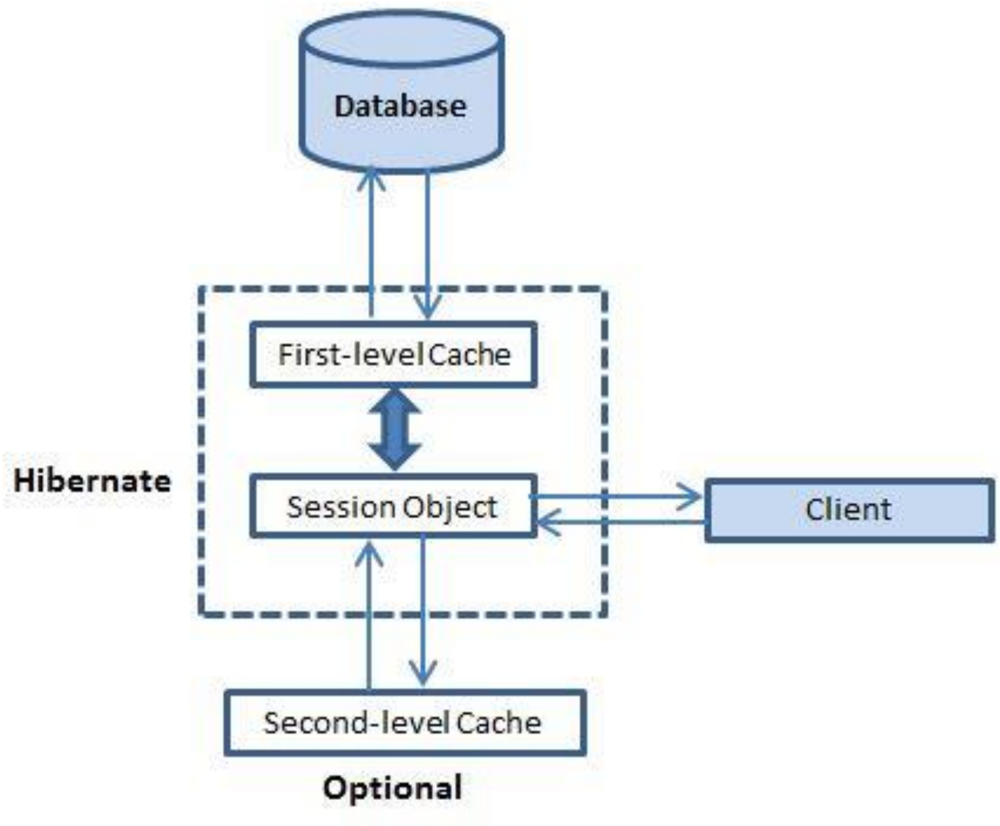


V. Liferay Tuning - Architecture



Layers from the perspective of code	
Layer 1	Liferay Thread Local Cache Only within current request scope
Layer 2	Entity (Pk) / Finder (Query Criteria) Cache -- ehcache (worker thread maintains own copy of data) System wide synchronize point Finder cache is built on top of entity cache. But the user sees finder and entity as the same.
Layer 3	Hibernate 2nd layer cache (Usually disabled - a duplication of our entity cache)
Layer 4	Hibernate 1st layer cache
Layer 5	Database

V. Liferay Tuning - Architecture



V. Liferay Tuning - Architecture

- All ServiceBuilder generated services can automatically leverage Liferay's L1 cache.
- Automatic clearing of all Liferay's L1 caches
- Cache via AOP using ThreadLocalCachable method annotation:

```
@ThreadLocalCachable
public Group getGroup(long groupId) { … }
```

V. Liferay Tuning - Architecture

- Within Liferay's L2 and L3 cache is an Entity and Finder/Query cache.
  - The entity level cache stores a value object’s primary key to the value object itself.
  - The finder level cache only caches primary keys.
  - By default, Liferay uses ehcache to manage these objects in memory.

V. Liferay Tuning - Architecture

- Liferay's L2 and L3 cache can be implemented as needed.
  - Determine a unique cacheName
  - Generate a unique objectKey
  - Update the elements within your LocalServiceImpl
- `Object value = SingleVMPoolUtil.get(cacheName, ObjectKey);`
- `Object value = MultiVMPoolUtil.get(cacheName, ObjectKey);`
- Best implemented after profiling finds a bottleneck in a particular area.

V. Liferay Tuning - Configuration

- Portal-ext.properties
  - This file has user customizations for Liferay.
  - It overrides the default settings in Portal.properties
  - It also overrides developer settings in Portal-developer.properties
  - A person can change settings such as whether to use DBCP or Tomcat threadpool.

V. Liferay Tuning - Configuration

- Clusterlink tuning
  - By default, cluster-link configuration uses RMI for L2 caching and UDP Multicast for other Liferay messages.
    - Cluster.link.enabled=true
  - An optimization is available to use UDP Multicast for L2 caching as well.
    - Ehcache.cluster.link.replication.enabled=true

V. Liferay Tuning - Configuration

- Deactivate Servlet filters you don't need:
  - AuditFilter, MonitoringFilter, CASFilter, NTLMFilter, NTLMPostFilter, OpenSSOFilter, SharepointFilter
- Deactivate in portal.properties (or portal-ext.properties)
- Prior to Liferay 6.1, you could also comment it out in web.xml

V. Liferay Tuning - Configuration

- Default configuration in portal.properties is set to optimize performance
- Portal-developer.properties makes life easier on developers (but can be removed in production for performance)
  - Theme.css.fast.load=false
  - Theme.images.fast.load=false
  - Javascript.fast.load=false
  - Combo.check.timestamp=true

V. Liferay Tuning - Configuration

- DB Query Optimizations
  - jdbc.default.idleConnectionTestPeriod=0
  - jdbc.default.preferredTestQuery=SELECT releaseId FROM Release_
  - jdbc.default.liferay.pool.provider=tomcat
- Search portal.properties on “performance” for other tips

V. Liferay Tuning - Configuration

- Remove unneeded portlets
  - Liferay comes pre-bundled with many portlets which contain a lot of functionality, but not every web site that is running on Liferay needs to use them all. In portlet.xml and liferay-portlet.xml, comment out the ones you are not using.

V. Liferay Tuning - Configuration

- Monitor Application Server Threads
  - Do not rely upon "auto-sizing" could result in "auto-thrashing"
  - Fast transactions imply 50-75 threads.
  - No more than 200-300 threads.
- Monitor JDBC connections
  - Initially size for 20 connections.
  - Adjust according to monitored usage

V. Liferay Tuning - Configuration

- Search optimization choice
  - Built-in Lucene search engine
  - SOLR web search engine plugin
  - Can configure the use of a SAN with Lucene or with SOLR

V. Liferay Tuning - Configuration

- CDNs Replicate content to servers closer to end user to reduce latencies
- Load static JS, images, etc. outside of Portal Application Server
- Reduces load on application servers

- Enable with *cdn.host.http* and *cdn.host.https*

V. Liferay Tuning – Code Changes

- Hooks
  - A hook contains one or more files that can overwrite files in the Liferay source portal-web folder.
- Patches
  - A patch is a file that contains one or more class files that will overwrite the original class files within Liferay.
- Ext's
  - An Ext is similar to a patch except that it is legacy and isn't compatible with the latest Liferay EE patching tool.

V. Liferay Tuning – Code Changes

- StringBuilder has worse memory performance than String.concat() when concatenating 2 or 3 strings.
- String.concat() has very slow CPU performance, compared to StringBuilder, for large numbers of strings.
- StringBundler has better memory performance than StringBuilder when concatenating large quantities of strings.

VI. Profiling and Monitoring Tools

VI. Profiling and Monitoring Tools

- Profiler-driven program analysis on Unix dates back to at least 1979, when Unix systems included a basic tool "**prof**" that listed each function and how much of program execution time it used. In 1982, **gprof** extended the concept to a complete call graph analysis.
- In 1994, Amitabh Srivastava and Alan Eustace of Digital Equipment Corporation published a paper describing **ATOM**. **ATOM** is a platform for converting a program into its own profiler. That is, at compile time, it inserts code into the program to be analyzed. That inserted code outputs analysis data. This technique - modifying a program to analyze itself - is known as "**instrumentation**".

VI. Profiling and Monitoring Tools

- Jmap
  - jmap prints shared object memory maps or heap memory details of a given process or core file or a remote debug server.
- Jstack
  - jstack prints Java stack traces of Java threads for a given Java process or core file or a remote debug server.
- Jconsole
  - The JConsole graphical user interface is a monitoring tool that complies to the Java Management Extensions (JMX) specification. JConsole uses the extensive instrumentation of the Java Virtual Machine (Java VM) to provide information about the performance and resource consumption of

VI. Profiling and Monitoring Tools

- Jstat
  - The jstat tool displays performance statistics for an instrumented HotSpot Java virtual machine (JVM).
  - jstat does not provide only the GC operation information display. It also provides class loader operation information or Just-in-Time compiler operation information.

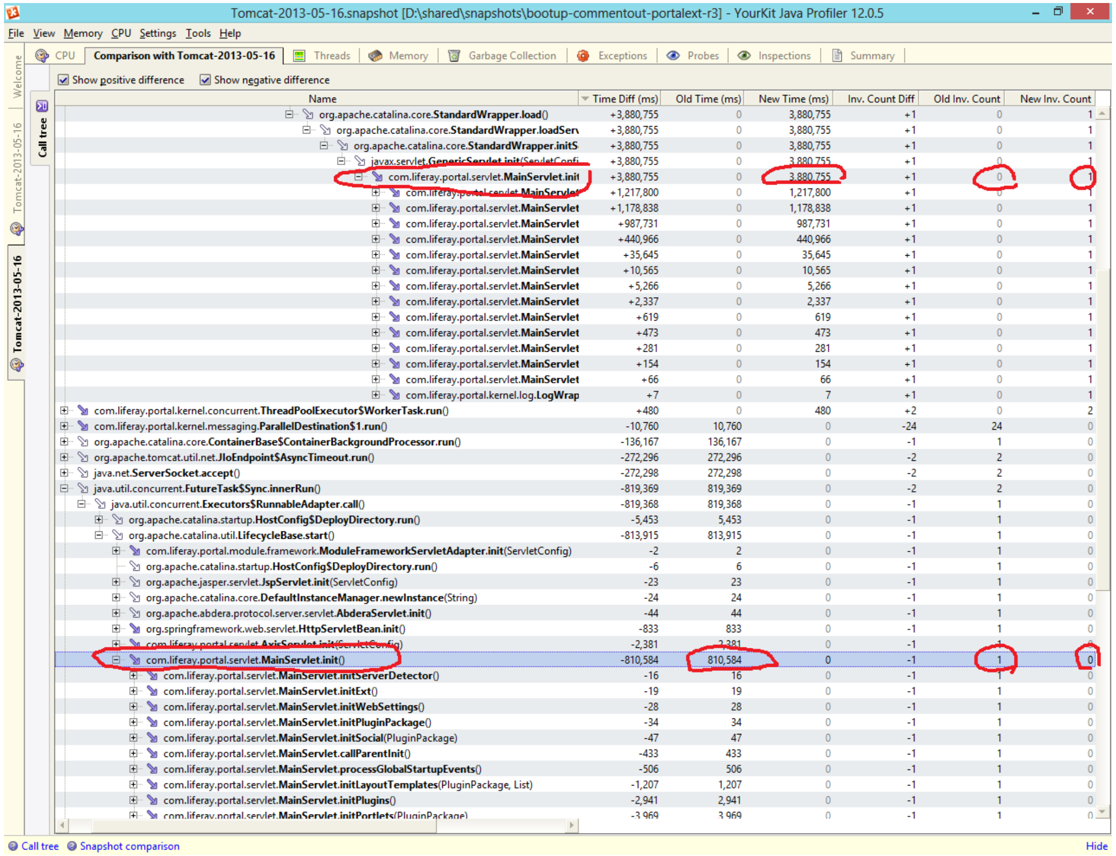
VI. Profiling and Monitoring Tools

- Monitor the CPU and virtual memory utilization
  - Vmstat – CPU and memory utilization
  - Mpstat – performance of each core/thread
- Monitor network and disk IO
  - Iostat – disk utilization and IO performance
  - Ifstat – network interface performance

VI. Profiling and Monitoring Tools

- VisualVM
  - “VisualVM is a visual tool integrating several commandline JDK tools and lightweight profiling capabilities.”
    - http://visualvm.java.net/
- Yourkit
  - “YourKit is a technology leader, creator of the most innovative and intelligent tools for profiling Java & .NET applications.”
    - http://www.yourkit.com/

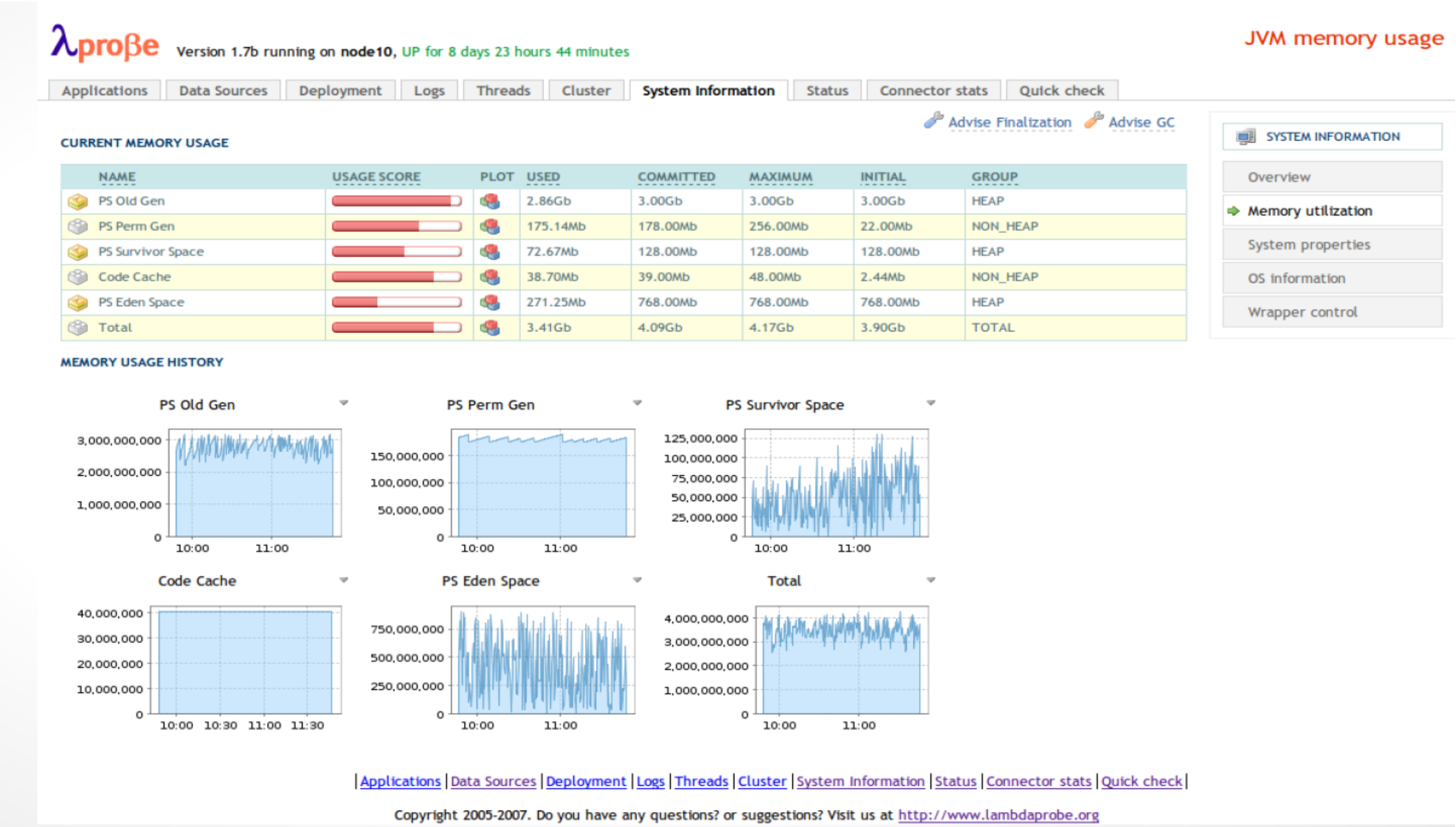
VI. Profiling and Monitoring Tools



☑ Show positive difference ☑ Show negative difference

Name	Time Diff (ms)	Old Time (ms)	New Time (ms)	Inv. Count Diff	Old Inv. Count	New Inv. Count
org.apache.catalina.core.StandardWrapper.load()	+3,880,755	0	3,880,755	+1	0	1
org.apache.catalina.core.StandardWrapper.loadServ	+3,880,755	0	3,880,755	+1	0	1
org.apache.catalina.core.StandardWrapper.initS	+3,880,755	0	3,880,755	+1	0	1
javax.servlet.GenericServlet.init(ServletConfi	+3,880,755	0	3,880,755	+1	0	1
com.liferay.portal.servlet.MainServlet.init	+3,880,755	0	3,880,755	+1	0	1
com.liferay.portal.servlet.MainServlet	+1,217,800	0	1,217,800	+1	0	1
com.liferay.portal.servlet.MainServlet	+1,178,838	0	1,178,838	+1	0	1
com.liferay.portal.servlet.MainServlet	+987,731	0	987,731	+1	0	1
com.liferay.portal.servlet.MainServlet	+440,966	0	440,966	+1	0	1
com.liferay.portal.servlet.MainServlet	+35,645	0	35,645	+1	0	1
com.liferay.portal.servlet.MainServlet	+10,565	0	10,565	+1	0	1
com.liferay.portal.servlet.MainServlet	+5,266	0	5,266	+1	0	1
com.liferay.portal.servlet.MainServlet	+2,337	0	2,337	+1	0	1
com.liferay.portal.servlet.MainServlet	+619	0	619	+1	0	1
com.liferay.portal.servlet.MainServlet	+473	0	473	+1	0	1
com.liferay.portal.servlet.MainServlet	+281	0	281	+1	0	1
com.liferay.portal.servlet.MainServlet	+154	0	154	+1	0	1
com.liferay.portal.servlet.MainServlet	+66	0	66	+1	0	1
com.liferay.portal.kernel.log.LogWrap	+7	0	7	+1	0	1
com.liferay.portal.kernel.concurrent.ThreadPoolExecutor$WorkerTask.run()	+480	0	480	+2	0	2
com.liferay.portal.kernel.messaging.ParallelDestination$1.run()	-10,760	10,760	0	-24	24	0
org.apache.catalina.core.ContainerBase$ContainerBackgroundProcessor.run()	-136,167	136,167	0	-1	1	0
org.apache.tomcat.util.net.JIoEndpoint$AsyncTimeout.run()	-272,296	272,296	0	-2	2	0
java.net.ServerSocket.accept()	-272,298	272,298	0	-2	2	0
java.util.concurrent.FutureTask$Sync.innerRun()	-819,369	819,369	0	-2	2	0
java.util.concurrent.Executors$RunnableAdapter.call()	-819,368	819,368	0	-1	1	0
org.apache.catalina.startup.HostConfig$DeployDirectory.run()	-5,453	5,453	0	-1	1	0
org.apache.catalina.util.LifecycleBase.start()	-813,915	813,915	0	-1	1	0
com.liferay.portal.module.framework.ModuleFrameworkServletAdapter.init(ServletConfig)	-2	2	0	-1	1	0
org.apache.catalina.startup.HostConfig$DeployDirectory.run()	-6	6	0	-1	1	0
org.apache.jasper.servlet.JspServlet.init(ServletConfig)	-23	23	0	-1	1	0
org.apache.catalina.core.DefaultInstanceManager.newInstance(String)	-24	24	0	-1	1	0
org.apache.abdera.protocol.server.servlet.AbderaServlet.init()	-44	44	0	-1	1	0
org.springframework.web.servlet.HttpServletBean.init()	-833	833	0	-1	1	0
com.liferay.portal.servlet.AxisServlet.init(ServletConfig)	-2,381	2,381	0	-1	1	0
com.liferay.portal.servlet.MainServlet.init()	-810,584	810,584	0	-1	1	0
com.liferay.portal.servlet.MainServlet.initServerDetector()	-16	16	0	-1	1	0
com.liferay.portal.servlet.MainServlet.initExt()	-19	19	0	-1	1	0
com.liferay.portal.servlet.MainServlet.initWebSettings()	-28	28	0	-1	1	0
com.liferay.portal.servlet.MainServlet.initPluginPackage()	-34	34	0	-1	1	0
com.liferay.portal.servlet.MainServlet.initSocial(PluginPackage)	-47	47	0	-1	1	0
com.liferay.portal.servlet.MainServlet.callParentInit()	-433	433	0	-1	1	0
com.liferay.portal.servlet.MainServlet.processGlobalStartupEvents()	-506	506	0	-1	1	0
com.liferay.portal.servlet.MainServlet.initLayoutTemplates(PluginPackage, List)	-1,207	1,207	0	-1	1	0
com.liferay.portal.servlet.MainServlet.initPlugins()	-2,941	2,941	0	-1	1	0
com.liferay.portal.servlet.MainServlet.initPortlets(PluginPackage)	-3,969	3,969	0	-1	1	0

VI. Profiling and Monitoring Tools



λproβe Version 1.7b running on **node10**, UP for 8 days 23 hours 44 minutes

JVM memory usage

CURRENT MEMORY USAGE

NAME	USAGE SCORE	PLOT	USED	COMMITTED	MAXIMUM	INITIAL	GROUP
PS Old Gen			2.86Gb	3.00Gb	3.00Gb	3.00Gb	HEAP
PS Perm Gen			175.14Mb	178.00Mb	256.00Mb	22.00Mb	NON_HEAP
PS Survivor Space			72.67Mb	128.00Mb	128.00Mb	128.00Mb	HEAP
Code Cache			38.70Mb	39.00Mb	48.00Mb	2.44Mb	NON_HEAP
PS Eden Space			271.25Mb	768.00Mb	768.00Mb	768.00Mb	HEAP
Total			3.41Gb	4.09Gb	4.17Gb	3.90Gb	TOTAL

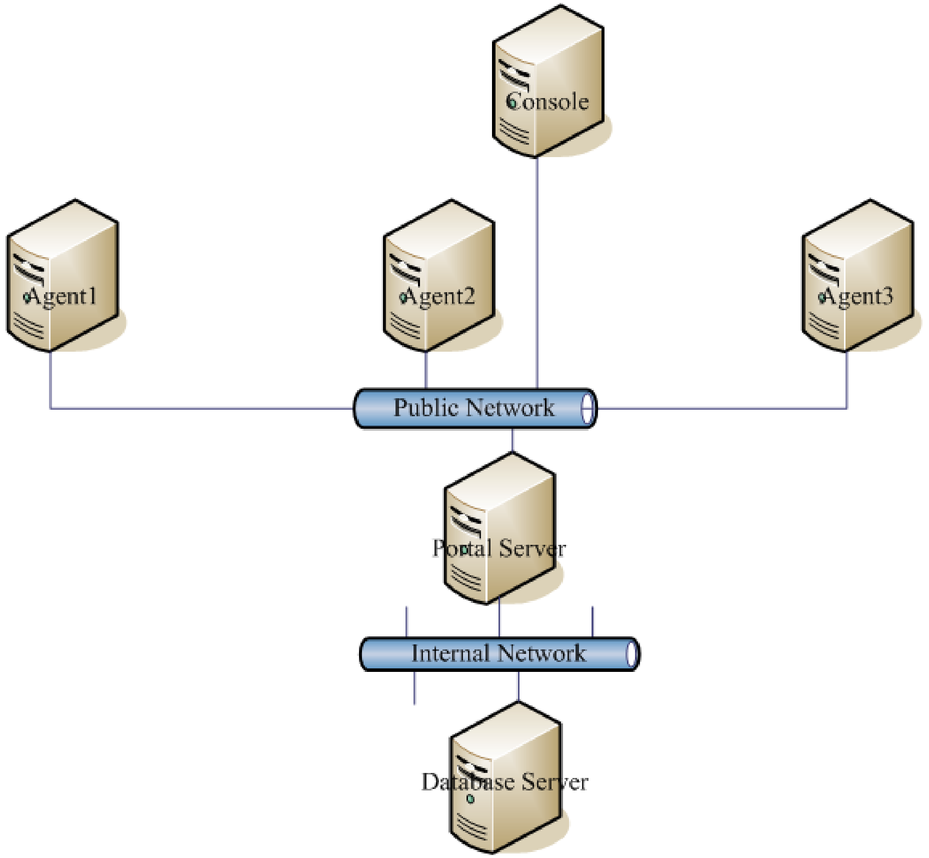
MEMORY USAGE HISTORY

|Applications|Data Sources|Deployment|Logs|Threads|Cluster|System Information|Status|Connector stats|Quick check|

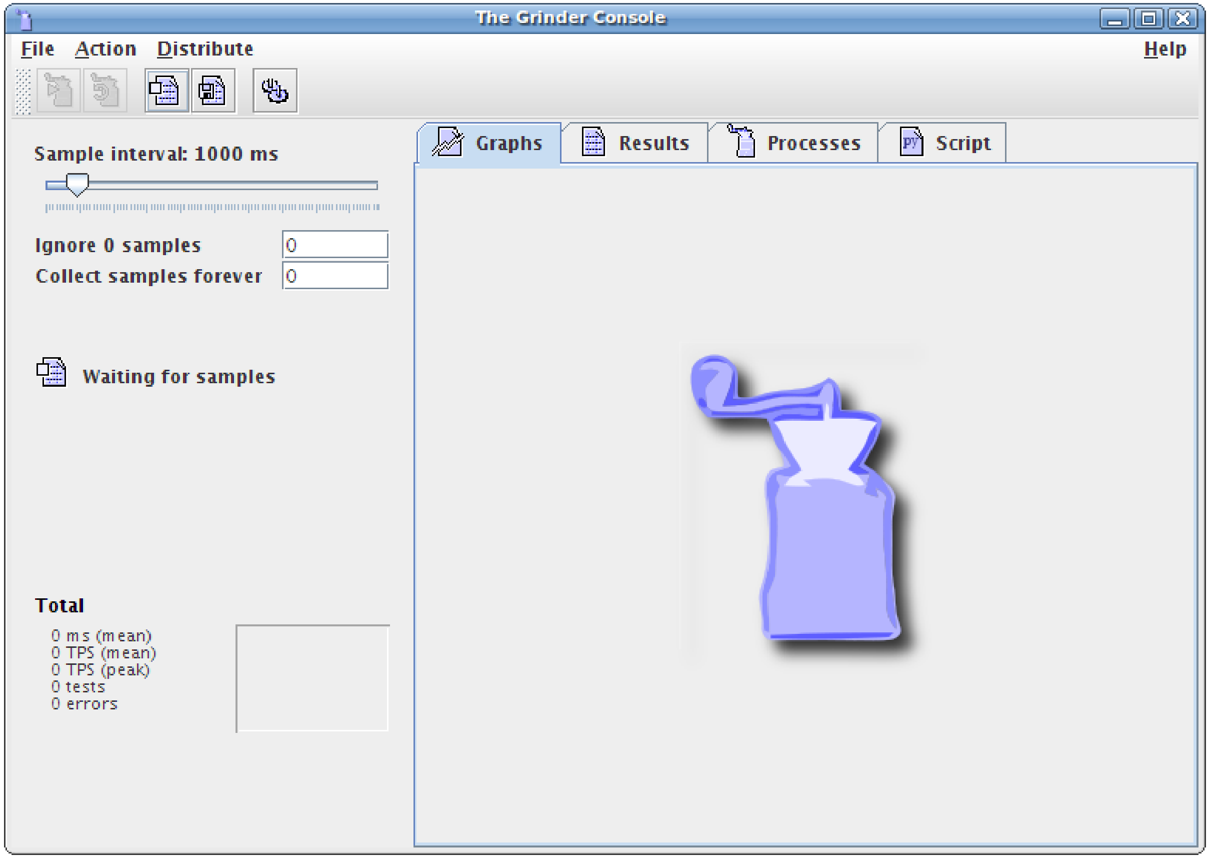
Copyright 2005-2007. Do you have any questions? or suggestions? Visit us at http://www.lambdaprobe.org

VII. Setting up a Performance Test

VII. Setting up a Performance Test



VII. Setting up a Performance Test



Sample interval: 1000 ms

Ignore 0 samples: 0

Collect samples forever: 0

Waiting for samples

Total

0 ms (mean)
0 TPS (mean)
0 TPS (peak)
0 tests
0 errors

VII. Setting up a Performance Test

- Simulating a live HttpSession
  - A sequence of HTTP URLs are requested with intermittent "sleeping" time in between each request.
  - Sleeping time is used to simulate a real user scenario.
  - The entire sequence begins with login(), ends with logout(), and is considered a single session.
  - Performance can then be measured in the total # of concurrent sessions.

VII. Setting up a Performance Test

- Concurrent Sessions vs. TPS
  - TPS measures how many URLs are being processed at a particular moment in time.
  - Concurrent Sessions measures how many sessions are live (requesting or sleeping) at a particular moment in time.
  - Sleeping sessions do not use CPU overhead, but do use system resources.

VII. Setting up a Performance Test

- Queuing Theory
  - Queue Length (X units) =

    Average arriving rate (Y units/second) *

    Average serving time (Z seconds)
  - Concurrent Session Count = TPS * Iteration Time

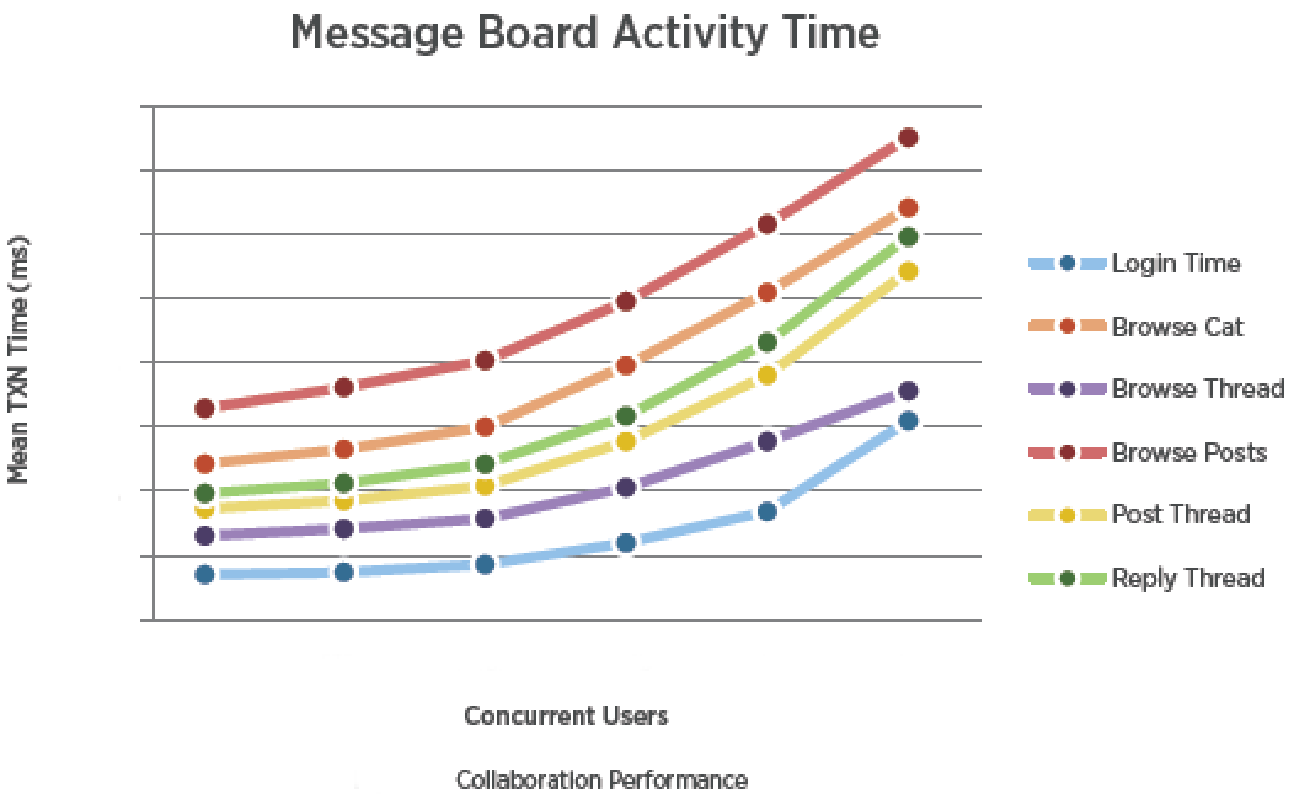
VII. Setting up a Performance Test

- Timing issues
  - Unless you set up a time server to provide timestamps to all the machines in the network, it would be best to have times measured within a specific machine and not across machines.

VII. Setting up a Performance Test

- Variable isolation
  - Commonly, when performance testing with Grinder, you will want to keep all variables in your system constant while you test different data points of a specific variable.

VII. Setting up a Performance Test



VII. Setting up a Performance Test

- What is the best, worst, and average case scenarios?
- What are the system limits?
- How does the software behave before, at, and after the system limits?

Summary

I. Definitions

II. Architecture

III. Requirements and Design

IV. JDK Tuning

V. Liferay Tuning

VI. Profiling and Monitoring Tools

VII. Setting up a Performance Test

Resources

- "Java Performance Tuning (2nd Ed)" by Jack Shirazi
- http://www.cubrid.org/blog/dev-platform/the-principles-of-java-application-performance-tuning/
- http://java.dzone.com/articles/java-performance-tuning
- http://www.liferay.com/documents/14/8440800/Advanced+Liferay+Architecture-Clustering+and+High+Availability.pdf
- http://www.slideshare.net/rivetlogic/liferay-developer-best-practices
- http://www.yourkit.com/docs/java/help/index.jsp
- http://www.oracle.com/technetwork/java/whitepaper-135217.html
- http://en.wikipedia.org/wiki/Performance_tuning