Advanced Liferay Architecture: *Clustering and High Availability*

Revision 1.1, Oct 2010

*Note: All of the configuration examples in 3rd-party software (i.e. – Apache, Sun Java) in this document are examples only. Please read the appropriate documentation and apply the settings that are appropriate for your system. Liferay does not recommend these exact same settings for all systems.

Contents

Overview

Once a standard Liferay installation has been completed, you may find that there is a need for a more advanced setup. You may need to prepare to handle more requests and arrange for more guaranteed uptime. This paper provides valuable insight and recommendations for configuring Liferay Portal to work in high availability (HA) use case scenarios.

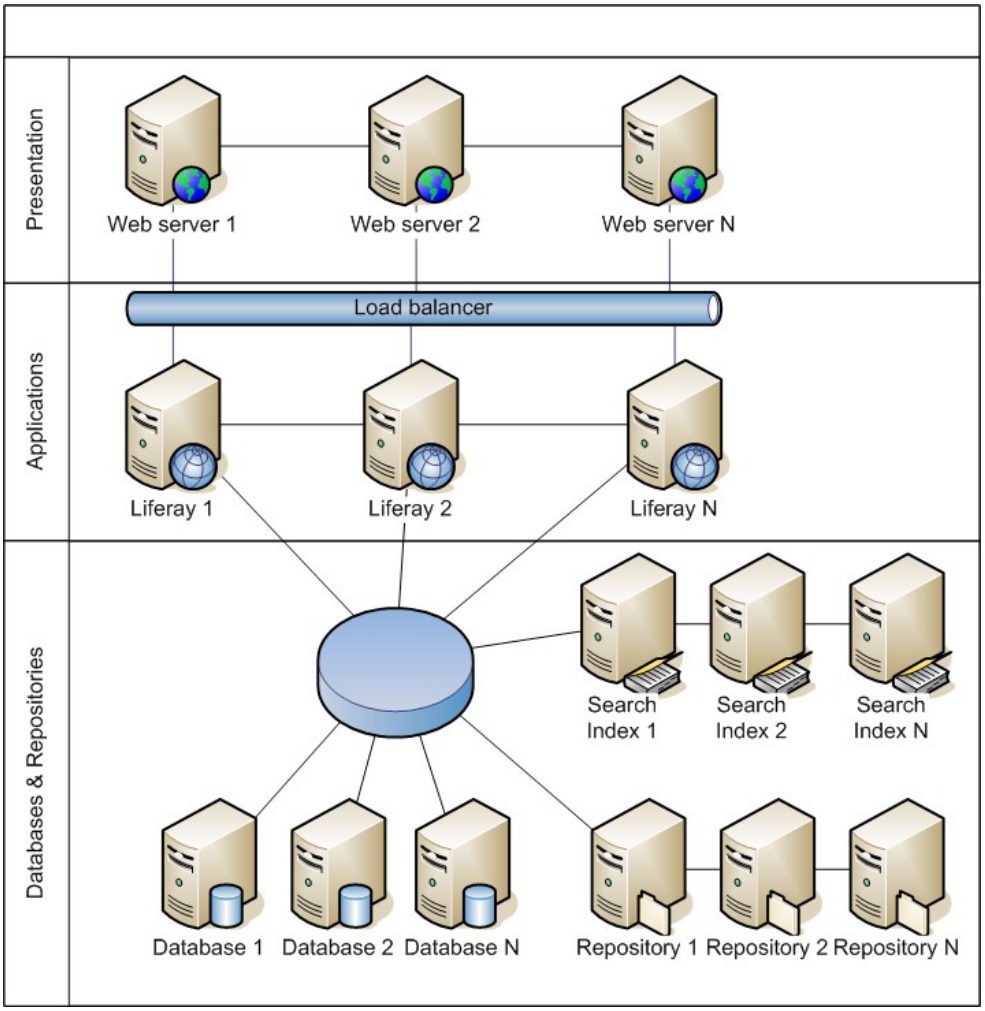
Is Clustering Enough?

HA is more than just server clustering, though that is part of the equation. It is a configuration that: 1) is able to handle the expected number of concurrent users and subsequent traffic, and 2) reduces single points of failure for a robust system uptime.

It is important to remember that Liferay Portal, at the end of the day, is a Java web application, and that many of the principles that apply to clustering any other Java web application running on your Java application server or Java servlet container apply here. We will be using familiar techniques such as load balancing and clustering at the server level, as well as Liferay-specific configuration settings for application-level HA.

It is important for us to define what "load balancing" is. Load balancing is simply a technique to distribute workload across resources. It is but one component in a high-availability cluster. In Liferay Portal's case, we are load balancing the volume of requests across multiple app servers, which may or may not be on physically separate hardware. Initially, this may seem sufficient, until you realize some of the components that the portal uses.

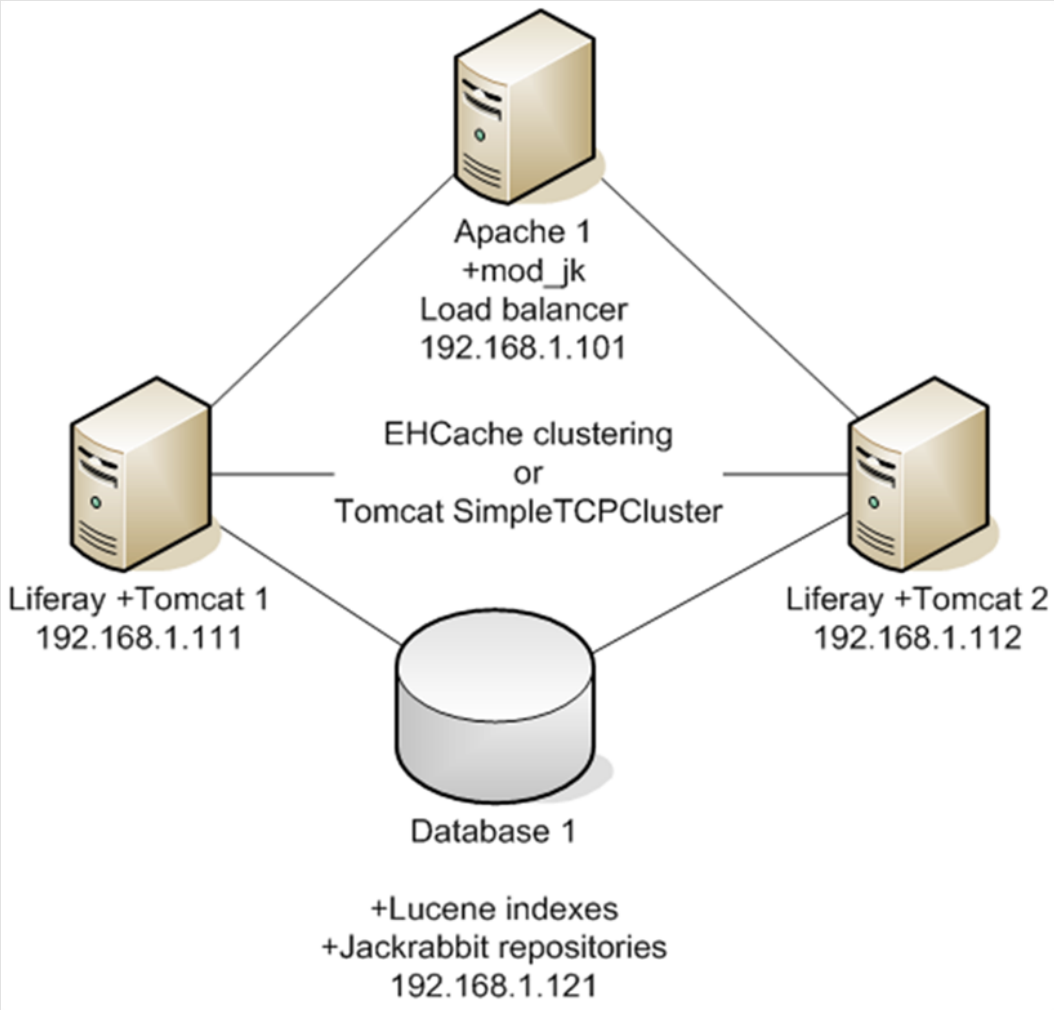
Here is a high-level picture of this type of set-up:



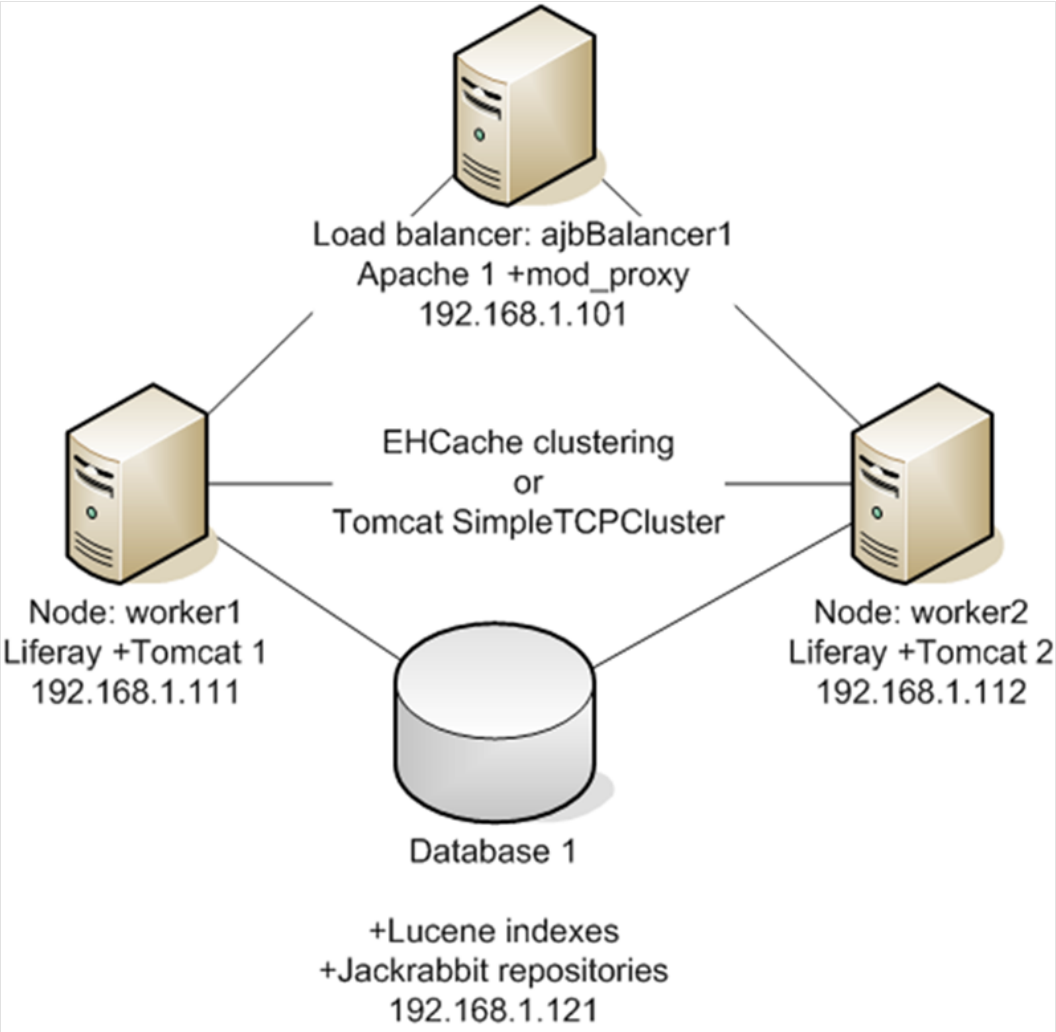
Sample Setups

The following are some sample configurations that you may want to consider. You may or may not have these exact configurations, or a derivative or permutation of one of them.

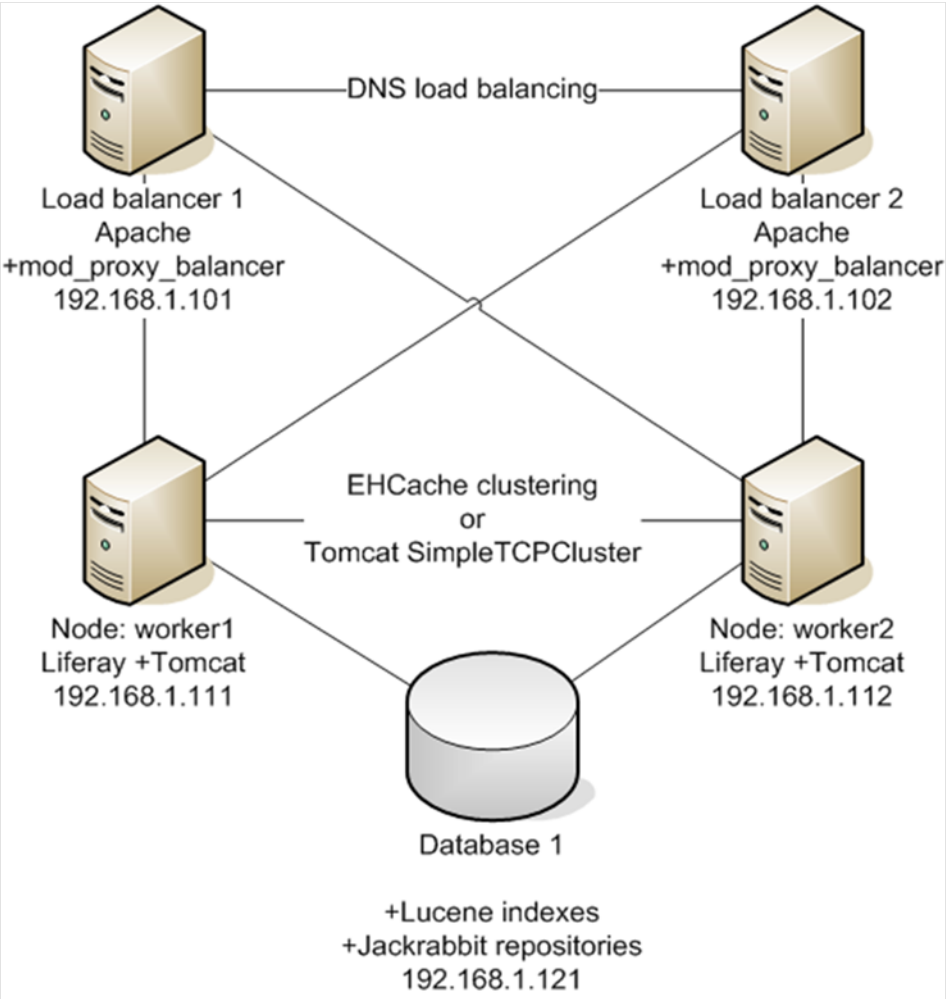
APACHE + MOD_JK + TOMCAT + LIFERAY:



APACHE + MOD_PROXY + TOMCAT + LIFERAY:



APACHE + MOD_PROXY_BALANCER + TOMCAT + LIFERAY:



What Is My Liferay Cluster Checklist?

SAMPLE CHECKLIST (LOGICAL VIEW):

Liferay clustering is not just pointing a load balancer to two (or more) Liferay nodes. You are not done. Why? Because there are certain application-level components that need to be either centrally managed or synchronized. Here is the basic checklist of components that need to be addressed in a Liferay cluster:

1. **Load Balancer** - This can be software (e.g., Apache), or hardware (e.g., F5), or whatever you wish, really--all it is doing is redirecting requests.
2. **Centralized Database** – The connection to the database is simply a JBDC connection. For the application server (or servlet container) it is a JNDI resource. Whether there exists a clustered database or not is abstracted from Liferay's point of view. Any level of redundancy you have behind that JDBC connection is up to you and your DBA. Just as an example, you may choose to configure a MySQL cluster, or Oracle RAC, for DB high availability. Also, we suggest that you move away from HSQL and use a more robust database server. HSQL is for demo purposes only.
3. **Ehcache** - This is what Liferay uses out-of-the-box for its Hibernate level 2 cache. This needs to be configured to sync or else you will see inconsistencies, depending on what node the load balancer redirects your end users to. You are not forced to use Ehcache; it is simply what Liferay ships with. You could use something like Terracotta, for example. If you do not do this, you will most definitely see inconsistencies depending on the node the end user is on, due to stale caches. For more information on how to get started with Liferay and Terracotta, please see: http://www.liferay.com/web/mika.koivisto/blog/-/blogs/how-do-i-cluster-liferay-with-terracotta.
4. **Lucene** - This needs to be centralized. This can be done a) via JDBC (can work, but there may be issues with speed and table locks); b) swapped out for something like SOLR (runs as a webapp in Tomcat); or c) a clusterlink feature (first made available with Liferay 5.2 SP2) that can be turned on where each node maintains its own replicated cache. If you do not do this, you will see inconsistencies with search and other indexed data that is returned from the DB.
5. **Document Library & Image Gallery** - This needs to be centralized. This is because each node keeps the assets locally on the file system by default. While the meta-data is in the DB, the files serve up faster this way (as opposed to BLOBS in a DB). So, you need to either a) point the content repository to be stored in the DB (can work but performance may suffer) via JCRHook in portal properties, or b) mount a path a shared storage (i.e. SAN or other file system storage) and configure each node to connect to this common place via *AdvancedFileSystemHook* in portal properties. If you do not do this, the meta-data regarding your documents will be in the DB, but when you try to retrieve them, the node may or may not find the document there, physically.

SAMPLE CHECKLIST (DETAILED FILE VIEW):

This is merely a sample of a list of files that may or may not need to be edited, or at least checked, for a proper Liferay HA cluster configuration. For other operating systems and application servers, please use the appropriate files on that system.

	Node	Applications	Operating System	Locations	Checklist		
Web Server	1	Apache 2.2.9 +mod_jk1.2.26	Debian 5.0 32bit	192.168.1.100	1	/etc/apache2/sites-available/default	√
					2	/etc/apache2/mods-enabled/jk.load	√
					3	/etc/apache2/ports.conf	√
					4	/etc/apache2/workers.properties	√
					5	/etc/hosts	√
					6	cat /proc/net/dev_mcast ifconfig -a | grep -i multicast	√
Liferay Portal	1	Liferay 6.0 GA +Tomcat 6.0.26	Debian 5.0 32bit	192.168.1.101	1	cat /etc/hosts	√
					2	cat /etc/profile	√
					3	echo $JAVA_HOME	√
					4	echo $PATH	√
					5	$TOMCAT_HOME/lib/portal-ext.properties	√
					6	$TOMCAT_HOME/conf/server.xml	√
					7	$TOMCAT_HOME/webapps/ROOT/WEB-INF/web.xml	√
					8	$TOMCAT_HOME/../data/jackrabbit/repository.xml	√
					9	$TOMCAT_HOME/webapps/ROOT/WEB-INF/classes/ehcache/hibernateclustered.xml	√
					10	$TOMCAT_HOME/webapps/ROOT/WEB-INF/classes/ehcache/liferay-multivm-clustered.xml	√
					11	cat /proc/net/dev_mcast ifconfig -a | grep -i multicast	√
	2	Liferay 6.0 GA +Tomcat 6.0.26	Debian 5.0 32bit	192.168.1.102	1	cat /etc/hosts	√
					2	cat /etc/profile	√
					3	echo $JAVA_HOME	√
					4	echo $PATH	√
					5	$TOMCAT_HOME/lib/portal-ext.properties	√
					6	$TOMCAT_HOME/conf/server.xml	√
					7	$TOMCAT_HOME/webapps/ROOT/WEB-INF/web.xml	√
					8	$TOMCAT_HOME/../data/jackrabbit/repository.xml	√
					9	$TOMCAT_HOME/webapps/ROOT/WEB-INF/classes/ehcache/hibernateclustered.xml	√
					10	$TOMCAT_HOME/webapps/ROOT/WEB-INF/classes/ehcache/liferay-multivm-clustered.xml	√
					11	cat /proc/net/dev_mcast ifconfig -a | grep -i multicast	√
Database	1	MySQL 5.0.51	Debian 5.0 32bit	192.168.1.103	1	/etc/mysql/my.cnf	√

A Quick Test To Check If Liferay Clustering Is Working

1. Bring up the load balancer
2. Bring up Node 1
3. Bring up Node 2
4. Open Browser 1 and go to Node 1 directly (i.e. – http://192.168.1.101:8080)
5. On Browser 1 go to page 1 (i.e. - http://192.168.1.101:8080/web/guest/home) logged in as Admin
6. Open Browser 2 (different browser session) and go to Node 2 directly (i.e. – http://192.168.1.102:8080)
7. On Browser 2 go to page 1 (i.e. - http://192.168.1.102:8080/web/guest/home) logged in as Admin
8. In Browser 1 add a portlet to the page
9. Shut down Node 1
10. Go to Browser 2 and refresh the page (i.e. – CTRL F5)

If Browser 2 refreshes and shows the added portlet from Browser 1, Liferay clustering in most likely configured correctly. This is because this behavior demonstrates that Node 2 is properly replicating cache. You can try reversing the order of Node 1 and Node 2 in the above test to be extra sure.

Performance Tuning Approach

For every performance tuning job, and there are some guiding principles that apply in every environment. A good place to start is to have a set of portal properties, xml configurations, and JVM flags that you know that you will use and apply every time. But that can only go so far, because every app is different and has different needs.

Performance tuning is a very iterative process. It involves 3 main components:

- A repeatable test script
- A load test client
- A Java profiler

GENERAL STEPS:

1. **Decide on what you want** - The first thing you need to do is to decide what you want to do for your test. Do you simply want to hit *localhost*, sign in, and sign out? Or do you want to browse around many pages with certain portlets placed on each page before signing out? Find out what the customer wants and decide on a test, or a number of test cases, early on and stay consistent throughout the performance tuning process. This will ensure meaningful results.
2. **Create a script** - The next thing you need to do is create the script for the repeatable test. For example, in Apache JMeter, it is easy to use the built-in proxy server to record a test script, based on your click actions. You can save this and run it multiple times across multiple threads.
3. **Decide on your numbers** - Decide how many concurrent users and repetitions you want to run this test as. For example, if the customer wants to test 200 concurrent users, you would configure JMeter to run 200 threads. You could maybe ramp them up 3 seconds apart, so in JMeter, you would configure the 200 threads to run within 600 seconds (i.e. - every 3 seconds, a thread will be started, each thread executing the test script).
4. **Establish a baseline** - This is a very important step, as this will prove that there was measurable improvement backed

up by statistics. In JMeter, you can use the aggregate report to gather data for this baseline. You have to show that you started somewhere.

5. **Use a Java profiler** - Why is this important? Because if you are stuck on why the CPU is spiking so much, why the memory keeps growing, why certain threads seem to be blocked, and so on. Apart from some very clever debugging in the code or a lucky guess, you will need a Java profiler to really nail down what the true cause is. This is because a Java profiler can examine what is in the JVM. It can examine what objects are in memory, how much memory you have available, how much memory is allocated, CPU performance and spiking, active threads, blocked or waiting threads, and much more. Recommended profilers are JProfiler, YourKit Java Profiler, and Netbeans Profiler.
6. **Identify bottlenecks** - Once you have the profiler in place, it will be much easier to see the bottlenecks. Is it too little heap? Is it processor power? Is there a memory leak? Are there blocked threads? Once you identify the bottlenecks, you can address those bottlenecks via configuration or code changes. Then deploy your changes.
7. **Repeat the above steps** - You must use the same test and run it the same way, after you have addressed your bottlenecks. You will most likely run into another, different bottleneck, but that is OK. The point is that you want to keep addressing the bottlenecks as they show up, address them, repeat the process, until you get the desired performance that you want. Take measurements and record statistics after every iteration! This will show progress.

If you do not use a Java profiler, it is almost always a wild guess as to what the problem is. Maybe an educated guess, but it's still a guess. Yes there is some overhead when attaching the Java profiler to the app server, but most times, this is negligible in testing scenarios. Again, this is very much an iterative process. Test, identify bottlenecks, address bottlenecks, rinse-repeat. Usually for customers we estimate 5 days. It's not something that can be done with quality, overnight. The iterations, configuration changes, code changes, redeployments, and testing all take time.

Lastly, do not run the tests in production! Make a copy and a test environment. The tests are supposed to be designed to slow down the system to reveal and identify the bottlenecks.

WHAT DO I TUNE?

When tuning your Liferay Portal, there are several factors to take into consideration, some specific to Liferay Portal, while others are concepts that apply to all Java and Java enterprise applications.

DISCLAIMER:
The following setting application server and JVM recommendations are only samples. The exact numbers that may be appropriate for your system may differ greatly from the samples presented here. Final configuration settings and numbers depend on the number of users your site needs to service, the level of responsiveness and fault tolerance that is acceptable to users, and the business requirements of your system. All of these factors can vary greatly from one Liferay deployment to another.

APPLICATION SERVER TUNING

Although how to change the settings may differ, the following parameters are applicable across most application servers. You should also consult your application server provider's documentation for additional specific settings that they may advise.

Deactivate Development Settings in the JSP Engine

Most application servers have their JSP Engine configured for development mode. Liferay recommends deactivating many of these settings prior to entering production:

- Development mode – this will enable the JSP container to poll the file system for changes to JSP files.
- Fork – fork the compilation of JSPs into a separate JVM. This is not required if development mode is off.
- Trim Spaces – removes excess whitespace between JSP Tag Library
- Mapped File – generates static content with one print statement versus one statement per line of JSP text.
- Generate String as Char Array – generates strings as static character arrays to relieve excess garbage collection.

Tomcat 5.5.x and 6.0.x

In the $CATALINA_HOME/conf/web.xml, update the JSP servlet to look like the following*:

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
      <param-name>fork</param-name>
      <param-value>false</param-value>
    </init-param>
    <init-param>
      <param-name>development</param-name>
      <param-value>false</param-value>
    </init-param>
    <init-param>
      <param-name>trimSpaces</param-name>
      <param-value>true</param-value>
```

```
        </init-param>
        <init-param>
            <param-name>mappedFile</param-name>
            <param-value>false</param-value>
        </init-param>
        <init-param>
            <param-name>genStrAsCharArray</param-name>
            <param-value>true</param-value>
        </init-param>
        <load-on-startup>3</load-on-startup>
    </servlet>
```

Thread Pool

Each incoming request to the application server consumes a thread for the duration of the request. When no threads are available to process requests, a request will be queued waiting for the next available thread. In a finely tuned system, the number of threads in the thread pool should be relatively limited. Configuring for too many threads will cause your machine to perform poorly, spending significant time switching between active threads.

Liferay Engineering recommends setting this initially to 50 threads and then monitoring it within your application server's monitoring consoles.

Tomcat 5.5.x and 6.0.x

In Tomcat, the thread pools are configured in the Connector element in $CATALINA_HOME/conf/server.xml. Further information can be found in the Apache Tomcat documentations (http://tomcat.apache.org/tomcat-6.0-doc/config/http.html). Liferay Engineering has used the following configuration during testing*:

```
<Connector port="8019" maxHttpHeaderSize="8192"
            maxThreads="50" minSpareThreads="50" maxSpareThreads="50"
            enableLookups="false" acceptCount="100" redirectPort="8443"
            protocol="AJP/1.3"
            connectionTimeout="20000" disableUploadTimeout="true" URIEncoding="UTF-8" />
```

Database Connection Pool

Set to roughly 20-30% of the thread pool size. The connection pool provides a connection whenever Liferay Portal needs to retrieve data from the database (e.g. user login, etc). If this size is too small, requests will queue in the server waiting for database connections. However, too large a setting will mean wasting resources with idle database connections.

Tomcat 5.5.x and 6.0.x

In Tomcat, the connection pools are configured in the Resource elements in $CATALINA_HOME/conf/ Catalina/localhost/ ROOT.xml. Liferay Engineering used the following configuration during testing*:

```
<Resource
auth="Container"
        description="Portal DB Connection"
        driverClass="com.mysql.jdbc.Driver"
        maxPoolSize="50"
        minPoolSize="10"
        acquireIncrement="5"
        name="jdbc/LiferayPool"
        user="XXXXXX"
        password="XXXXXXXXX"
        factory="org.apache.naming.factory.BeanFactory"
        type="com.mchange.v2.c3p0.ComboPooledDataSource"


        jdbcUrl="jdbc:mysql://someServer:3306/lportal?useUnicode=true&amp;characterEncoding=UTF-
8&amp;useFastDateParsing=false"/>
```

In this configuration, we have a maximum of 50 connections in the pool and starting with 10. Should all 10 be used, the pool will automatically add 5 additional connections.

JAVA VIRTUAL MACHINE (JVM) TUNING

Tuning the JVM primarily focuses on tuning the garbage collector and the Java memory heap. These parameters look to optimize the throughput of your application. We used the Sun 1.6 JVM for the reference architecture. You may also choose the IBM JVM and Oracle's JRockit JVM.

Garbage Collector

Choosing the appropriate garbage collector will help improve the responsiveness of your Liferay Portal. Liferay Engineering recommends using the concurrent low pause collectors, similar to the settings as follows*:

-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:ParallelGCThreads=8 -XX:+CMSCompactWhenClearAllSoftRefs -XX:CMSInitiatingOccupancyFraction=85

Other garbage collectors include a concurrent throughput collector. You may find more information on garbage collector heuristics at http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html.

Java Heap

When most people think about tuning the Java memory heap, they think of setting the maximum and minimum memory of the heap. Unfortunately, you require far more sophisticated tuning for the heap to obtain optimal performance, including tuning young generation size, tenuring durations, survivor spaces, and other JVM internals.

For the reference architecture, Liferay recommends adding something similar to the following to your VM parameters*:

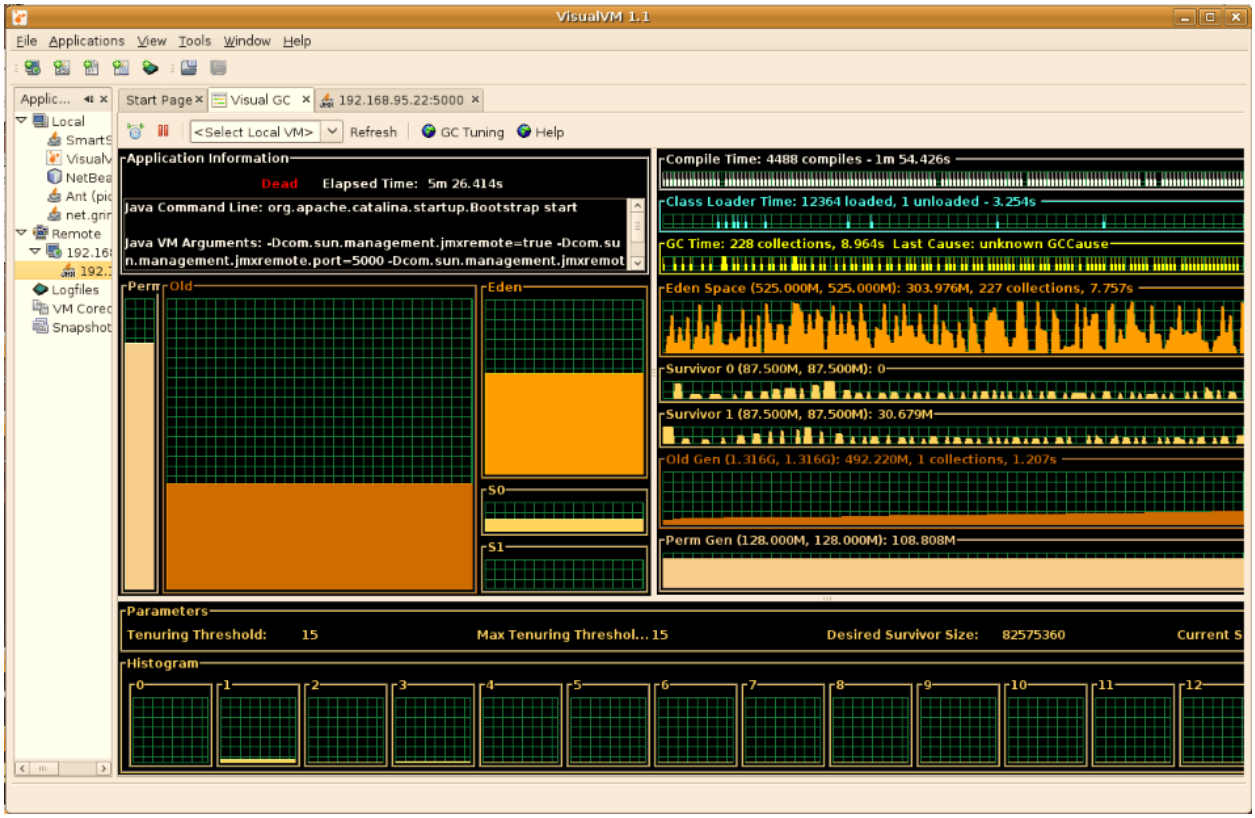
-server -XX:NewSize=700m -XX:MaxNewSize=700m -Xms2048m -Xmx2048m -XX:MaxPermSize=128m -XX:SurvivorRatio=6 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=15

You may find the definitive list of Sun JVM options are located at: http://blogs.sun.com/watt/resource/jvm-options-list.html.

Monitoring GC and JVM

Although the parameters introduced here will give you a good start to tuning your JVM, you should still monitor it to ensure you have the best settings to meet your needs. There are several tools to help you monitor Sun JVM performance including:

Visual VM (http://visualvm.dev.java.net)
This tool provides a centralized console for viewing Sun JVM performance information, including garbage collector activities.

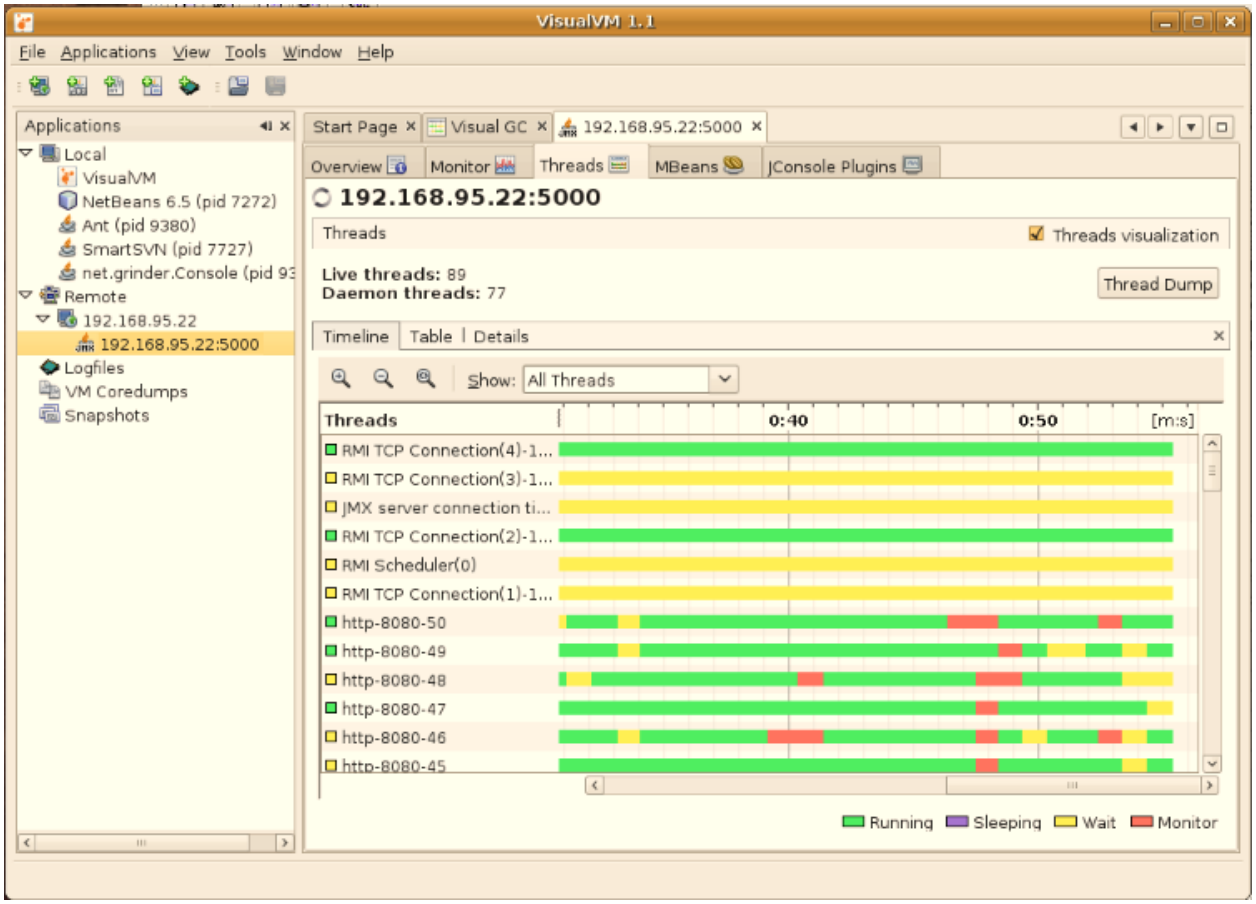


[Visual VM's Visual GC]

JMX Console

This tool helps display various statistics like Liferay's distributed cache performance, the performance of application server threads, JDBC connection pool usage, and etc.
You may add the following to you application server's JVM arguments to enable JMX connections*:

-Dcom.sun.management.jmxremote=true -Dcom.sun.management.jmxremote.port=5000 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false



[Visual VM's JMX Console]

Garbage collector logs

You may add something similar to the following to your JVM arguments*:

-Xloggc:/tmp/liferaygc1.log -XX:+PrintGCDetails -XX:+PrintGCApplicationConcurrentTime -XX:+PrintGCApplicationStoppedTime

SO WHAT ARE SOME EXAMPLE SETTINGS?

A lot of what goes into JVM tuning has to deal with memory management. For a very detailed reference on memory management in the Sun JVM, please see the Sun whitepaper, “Memory Management in the Java Hotspot Virtual Machine” at:

http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

Reference Hardware: 2 quad(4)-core CPU machine with 8 GB memory

What do we do with that?

1) Check your portal server's CPU and memory.

Our portal server has 2 4-core CPUs, which means 8 CPUs and 8GB physical memory for the JVM.

Because we have multiple CPUs, we should try to use a multi-threaded setting to speed up garbage collection (GC).
In the Sun JDK5, there are 4 built-in GC types: Serial Collector, Parallel Collector, Parallel Compacting Collector and Concurrent Mark-Sweep (CMS) Collector. For our server, we will choose Concurrent Mark-Sweep (CMS) Collector, because we have 8 CPUs and we need a short GC pause time. (For more detail, please read the Sun whitepaper, “Memory Management in the Java Hotspot Virtual Machine”). Use **-XX:+UseConcMarkSweepGC** to turn on this option.

2) Fix your Xms and Xmx to a same value.

Because our server is a dedicated server for Liferay Portal, once you know a suitable heap size for the whole JVM, there is no reason to define a range and let the JVM resize the heap. We should explicitly set the heap size to the suitable value. **Use -Xms2048m -Xmx2048m** to set the heap size. (We will explain why it is 2048 later.)

3) Set the young generation size (Don't know what the young generation is? Please read see the Sun whitepaper, “Memory Management in the Java Hotspot Virtual Machine”)**.**

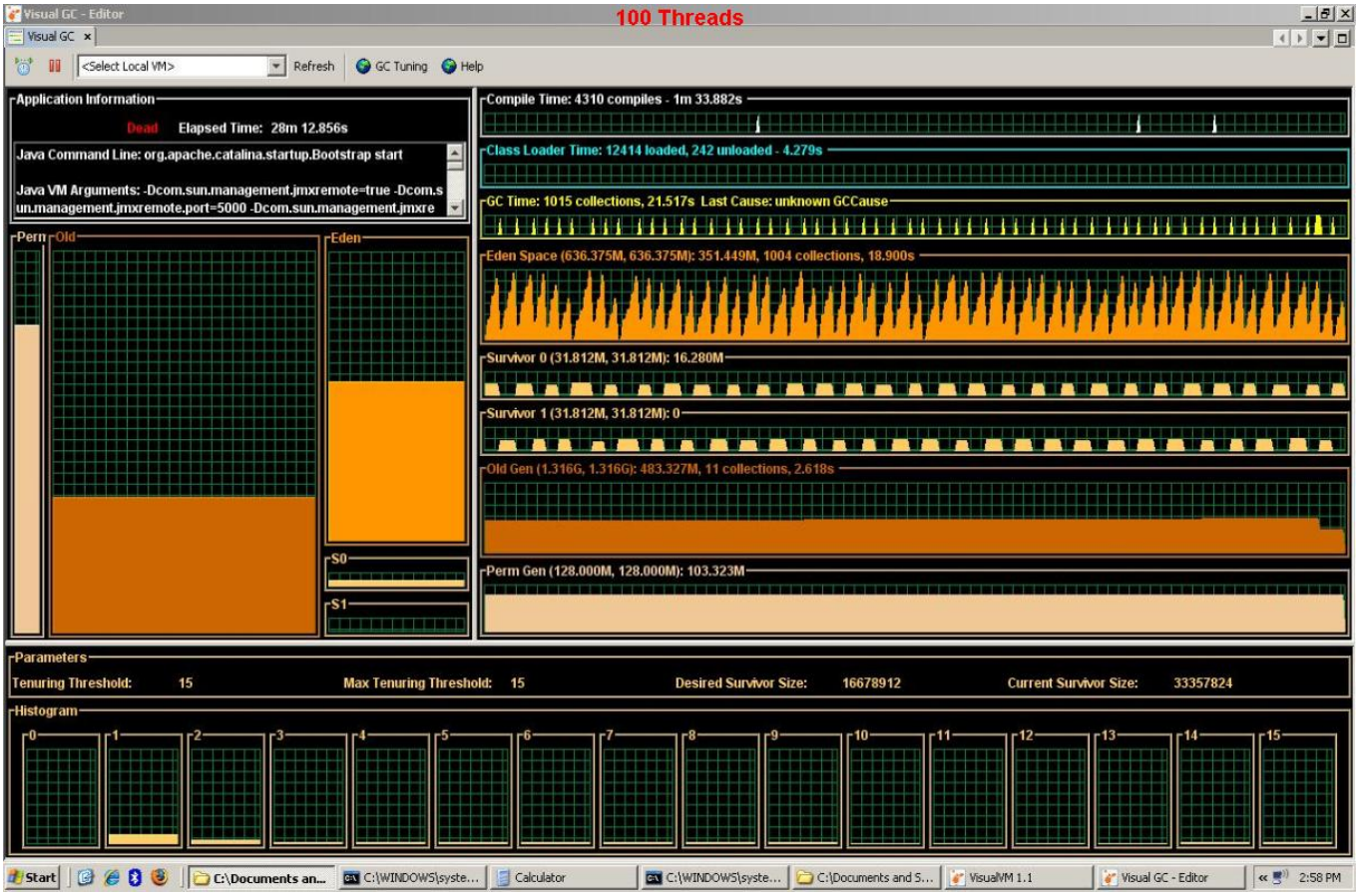
For the same reason as above, we should set the young generation size explicitly to a suitable size. Experimentally we will make the young generation size 1/3 of the whole heap. Use **-XX:NewSize=700m -XX:MaxNewSize=700m** to set the young generation size.

4) Set PermSize.

PermSpace is used to store class objects, so the size depends on how many classes do you have. In our case let’s just say that 128MB is big enough for now. If you get an OutOfMemory error complaining that there is no more space in PermSpace, you should increase this value. Use **-XX:MaxPermSize=128m** to set the PermSize.

5) Set the SurvivorRatio to young generation.

There is no particular rule for this, the value is derived from observing a Java profiler (i.e. - VisualVM). You should try to make survivor size bigger than the peak memory use size. Here we’ll set the ratio to 20. Use **-XX:SurvivorRatio=20** to set the ratio.



[Monitoring the JVM heap and garbage collection in Visual GC]

Now let us talk about why set the heap size to 2048MB. We are using a 64-bit JVM, and we are aware that it can support more than a 2048MB heap size, but that does not mean setting the heap size to the maximum. Bigger is not always better. We chose a heap size that is appropriate. The algorithm for GC tells us as the heap size increases linearly, the rate at which GC grows is actually faster than linear. So we should just set the heap size to meet our need. In our case, through testing in our Java profiling tool, VisualVM, we chose 2048MB for the heap size. From the statistics in the GC time for young generation and old generation, a suitable heap size is estimated. Generally the average young generation GC time should around 20ms, while the old generation GC time should around 200ms~400ms.

The final startup parameters should look something similar to this*:

JAVA_OPTS="$JAVA_OPTS -XX:NewSize=700m -XX:MaxNewSize=700m -Xms2048m -Xmx2048m -XX:MaxPermSize=128m -XX:+UseConcMarkSweepGC -XX:SurvivorRatio=10"

Additional Information

For more information on Liferay cluster settings, please see the Liferay Portal Administrator's Guide at:

http://docs.liferay.com/portal/5.2/official/liferay-administration-guide.pdf

Moving Forward

LIFERAY TRAINING

Liferay offers a Liferay System Administration course, where you can configure an actual Liferay cluster, hands-on from scratch. Please see http://www.liferay.com/services/training/topics/system-administrator-training for more information.

Contact training@liferay.com for more information.

LIFERAY ENTERPRISE EDITION SUPPORT

Liferay Enterprise Edition ensures stability and reliable technical support for your Liferay Portal installation and your organization's team., including a customer portal, product bulletins, security alerts, plus the support of over 60 partners worldwide.

LIFERAY PROFESSIONAL SERVICES

Liferay offers performance tuning and architecture assistance for your system. Since specific performance tuning settings may vary from system to system, Liferay recommends thorough evaluation of your environment and business requirements before applying any of the settings mentioned in this paper.

Contact sales@liferay.com for more information.

*Note: All of the configuration examples in 3rd-party software (i.e. – Apache, Sun Java) in this document are examples only. Please read the appropriate documentation and apply the settings that are appropriate for your system. Liferay does not recommend these exact same settings for all systems.