Performance Improvement of Jboss EAP 4.3 Platform

Ruchir Choudhry[#1], Alpesh Vaghela #[2], Vaidyanathan Kothandaraman[#3]

Web Architecture

Joseph Monsanto, Kaliki Jan
VM Architecture team

NBC Universal, 100 Universal City Plaza, CA, 91608-1002 USA
[1]ruchir.Choudhry@nbcuni.com
[2] alpesh.Vaghela@nbcuni.com
[3]vaidyanahan.kothandaraman@nbcuni.com
jan.kaliki@nbcuni.com
Joseph Monsanto@nbcuni.com

Abstract:
One of the principle objectives of this endeavour was to increase the performance of the overall system without touching the code and to formulate a strategy in which we can utilize the best practice provided by Apache, Red Hat Jboss, Sun JDK and various other distributed components. This white paper provides an in-depth knowledge of how to increase the performance of the system without going into the complexity of redoing the code.

I. INTRODUTON

During the last few years J2EE/JavaEE based applications have largely spread over our networks. Our Jboss EAP 4.3/ J2EE platform uses a distributed multi tiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components that make up a J2EE application are installed on different machines depending on the tier in the multi tiered like

- Client-tier components run on the client machine.
- Web-tier components run on the J2EE server.
- Business-tier components run on the J2EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Taking this distributed component into preview this white paper works as a baseline for Jboss EAP 4.3 J2EE/ Java EE platform in NBC Universal and provide guideline for the performance counters from the core infrastructure components, application server, webserver, java engine.

II. BEST PRACTICE

3) Use the most recent version of JDK /JRE

For each major Java™ release "train" (e.g.J2SE 1.4.2, J2SE 5.0, J2SE 6.0) Sun publishes update releases on a regular basis. For example the most recent update release of Java SE 1.6.0_18. Update releases often include bug fixes and performance improvements. By deploying the most recent update release for Java™ you will benefit from the latest and greatest performance improvements.

4) Insure OS patches are up-to-date

Even though Java is cross platform it does rely on the underlying operating system and therefore it is important that the OS basis for the Java™ Platform is as up-to-date as possible. In our case we used Red Hat Enterprise 5.3 on VM.

5) Use trimmed version of Web Server (Apache2.2.8, 2.2.15)[9][10], by compiling it specific for the needs of the applications or env.

Apache comes with various configuration parameters to give more and more flexibility to the application. It's very much required to pick what is good for us and what is the need of the application. Based on NBC/Ge needs we have trimmed the Apache modules, and recompiled for NBC requirements. The details are in the appendix 1

6) Place the code on the local env not on the storage; content can reside on the storage.

By placing the code on the local (a carved out space on virtual machine (VM Ware) from the storage), there is no traversing to the storage, rather is utilizes the higher capacity bandwidth, which is dedicated for the VM Ware env.

7) Trim the application server (Jboss EAP 4.3) by exactly using what you need from the application server

By trimming the Jboss EAP 4.3 we mean to remove a lot of services which comes with a J2EE container by default like JMS, Session based clustering, large size of object pooling, connection poking , XA Transaction, Modjk tomcat connectivity and much more the details are there in the appendix 2

8) Use logging as into

Use only info type of logging in production; this is done to reduce the I/O to the file system.

9) Test various JVM[1][2][3][13][14] configuration settings and make sure to use what is best for the en, appendix1

We used this to find out what is the right size for our env and how it fluctuates based on the different application needs. Based on our test, we found that 2 VCPU with 4 GB of RAM is best suited for our applications/env.

The details of the JVM parameters and their changes are in appendix 3

10) Test various CPU or VCPUS, RAM options to find out what's best suited for the env

The right size of the CPU & RAM varies due to type of hardware used; design an type of application, how the multithreaded env is used/programmed, type of platform and so on and so fourth. We used VM Ware RED HAT 5.3 Enterprise edition on ESX farms. We did various rounds of test, as detailed in appendix 4, and found that 2VCPU along with 4 GB of RAM per Jboss EAP4.3 is needed.

11) Use hardware based load balancing to provide a HA env, Avoid clustering if its not the business need

Its tempting to use software based clustering and it can be done at various levels and in various ways. But in all possible scenarios it creates an overhead.
Example:
HAPartition[15] is a general-purpose service used for a variety of tasks in AS clustering. At its core, it is an abstraction built on top of a Jgroups[15] Channel that provides support for making/receiving RPC invocations on/from one or more cluster members. HAPartition allows services that use it to share a single Channel and multiplex RPC invocations over it, eliminating the configuration complexity and runtime overhead of having each service create its own Channel.

12) Disable hot deployment if its not needed in production

Looking at the hot deployment features supported by various vendors everybody thinks it's "cool idea" but not robust as it appears. None of the vendors recommend wholeheartedly these in production environment, so you have to be careful when you decide to use this in your production server running your website or mission critical applications. In addition to the above during hot deployment Container spends significant CPU cycles polling for changes in applications. This is the sole reason we decided not to use hot deployment.

13) Start Jboss EAP 4.3 always in production mode if the application is in production

During our series of test we did find the best startup mode among the various modes of startup (default, all, minimal and production) is production. This gave us the best possible performance.

14) Eliminating Variables

Be aware that various system activities and the operations of other applications running on your system can introduce significant variance into the measurements of any application's performance, including Java applications. The activities of the OS and other applications may introduce CPU, Memory, disk or network resource contention that may interfere with your measurements.

Before you begin to measure Jboss platform performance try to assess application behaviour that may *discolour* your performance results (e.g. accessing the Internet for updates, reading files from the users home directory, etc.). By simplifying application behaviour as much as possible and by changing only one variable at a time (i.e. operating system tuneable parameter, Java command line option, application argument, etc.) your performance investigation can track the impact of each change independently.

III. MAKING DECISION ON DATA

It's tempting to run an application once before and once after a change and draw some conclusion about the impact of that change. Sometimes the application runs for a long time. Sometimes launching the application may be complex and dependent on multiple external services. But can you legitimately make a decision from this test? It's really impossible to know if you can safely draw a conclusion from the data unless you measure the power of the data quantitatively. Applying the scientific method is important when designing any set of experiments. Rigor is especially necessary when measuring Java application platform's performance because the behaviour of Java Hotspot virtual machine adapts and reacts to the specific machine and specific application it is running. And subtle changes in timing due to other system activity can result in measurable differences in performance and which are unrelated to the comparisons being made.

Prior to the experiment we tried to eliminate as much application performance variability as possible. However it is **rarely possible to eliminate *all* variability,** especially noise from asynchronous operating system services. By repeating the same experiment over the course of several trials and averaging the results you effectively focus on the signal instead of the noise. The rate of improving the signal is proportional to the square root of the number of samples.

To put Java/Jboss benchmarking into statistics terms we are going to test the *baseline* settings before a change and the *specimen* settings after a change. For example you may run a benchmark baseline test 10 times with no Java/Jboss command line options and specimen test 10 times with the command line of "-server". This will give you two different sample populations. The questions you really want to answer are, "Did that change in Java settings make a difference? And, if so, how much of a difference?".

The second question, determining the percentage improvement is actually the easier question:
percentageImprovement = 100.0 x (SpecimenAvg - BaselineAvg) / BaselineAvg

The first question, "Did that change in Java/Jboss settings make a difference?" is the most important question, however because what we really want to know is "Is the difference significant enough that we can safely draw conclusions from it?" In statistics jargon this question could be rephrased as "Do these two sample populations reflect the same underlying population or not?" To answer this question we use the Students t-test. Using the number of samples, mean value and standard deviation of the baseline population and the specimen population as well as setting the desired risk level (*alpha*) we can determine the *p-value* or probability that the change in Java settings was significant. The risk level (*alpha*) is often set at 0.05 (or 0.01), which means that five times (one time) out of one hundred you would find a statistically significant difference between the means even if there were none. Generally speaking if the *p-value* is less than 0.05 then we would say that the difference is significant and thus we can draw the conclusion that the change in Java/Jboss settings did make a difference.

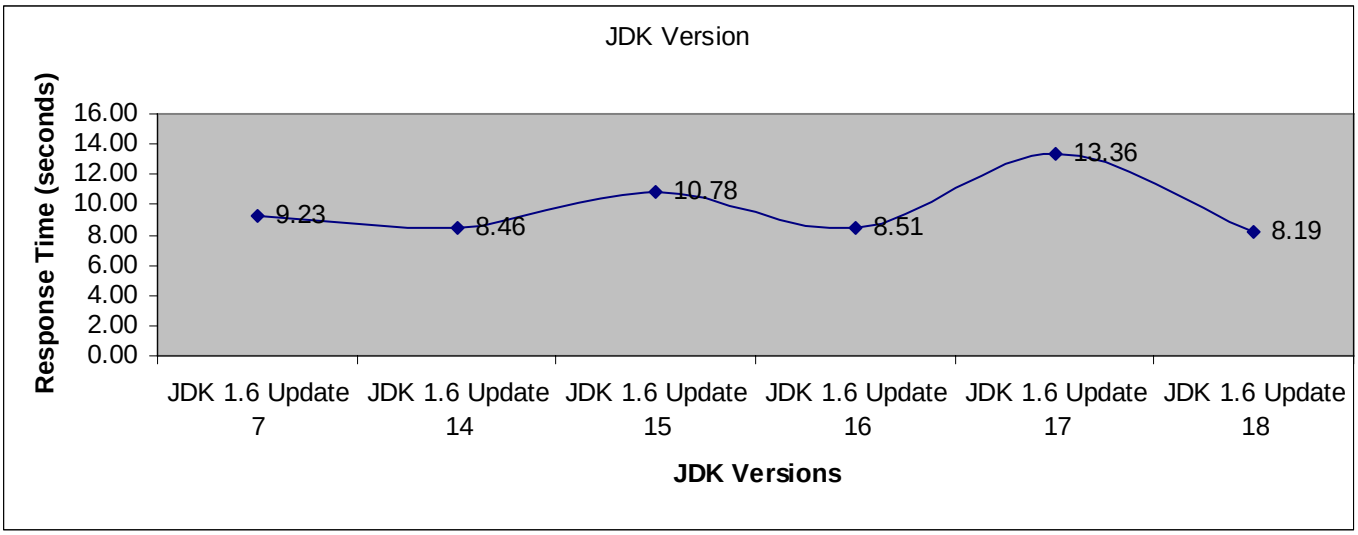
IV. GENERAL JBOSS EAP 4.3 TUNNING GUIDELINES

In the above section II of BEST PRACTICE we discussed about some of the fundamental aspects of the env tuning, we also found out the right way to find and analysis the data that is churned out due to various test which we performed. This section on Tuning Ideas contains suggestions on various tuning options that you should try on with your Jboss EAP 4.3 and Java application. All comparisons made between different sets of options are performed using the statistical techniques discussed above.

01) Selecting the right JDK and JRE env

Java platform is very unique and very versatile to get maximum out of it we must know which is the exact versions best for our need.

To quantify this we did various round of test with multiple applications in the same Jboss EAP4.3 J2EE container (Noise Test). This gives a much closer simulation of the actual production env. Based on our test we were able to find that Jdk1.6_18 is best suited for NBCU/GE env.
The details are depicted in the 01 –graph as below.



02) Selection of the right set of JVM parameter

Selecting a right size for the JVM for a J2EE/Java EE container, which holds multiple applications and providing the right size for every application is impossible as every application is unique based on it, JVM needs and can perform very differently under different condition. For our purpose we conducted series of test on various applications (Appendix 02), which is as follows.

- Single user test
- Noise test
- Endurance test
- Stress test

Based on these test results we can conclude that the case 09(details are there in the appendix 01) is the best suited for our env

-

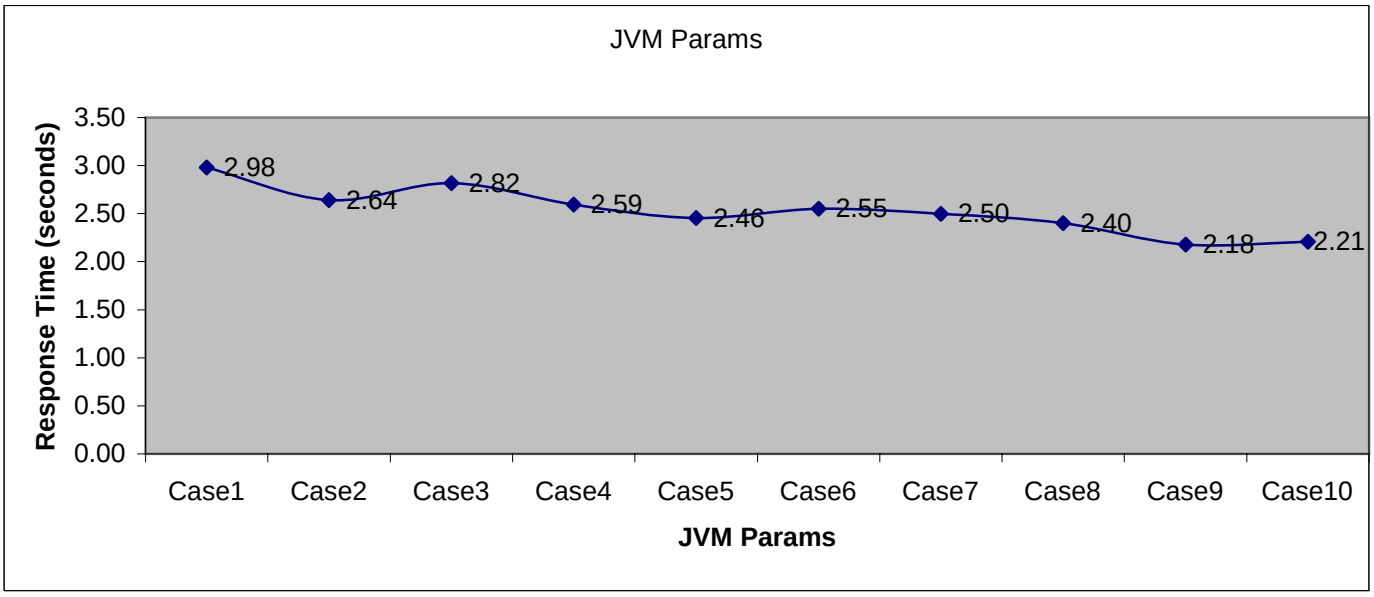
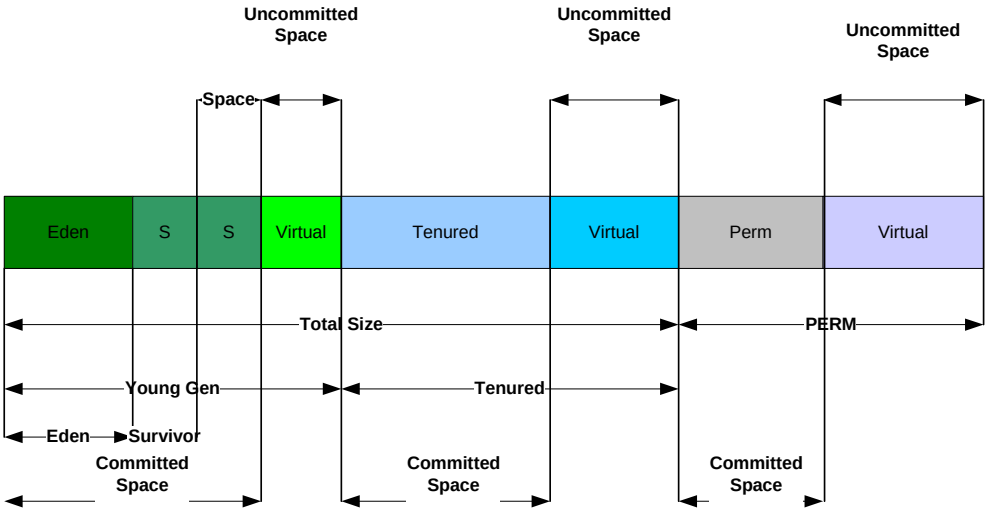


Figure – 02-JVM Params.




```
Details
--------
Young Gen: The young generation consists of eden plus two survivor spaces
           Objects are initially allocated in eden
           One survivor space is empty at any time, and serves as a destination of the next, copying collection of any live
            objects in eden and the other survivor space.
Tenured  : The old objects which is copied between the survivor space are copied to the tenured section.

Permanent
Generation
(Perm Gen): It is special because it holds data needed by the virtual machine to describe objects that
            do not have an equivalence at the Java language level. For example objects describing classes and methods are
            stored in the permanent generation.
Committed Space : This space is used as the JVM start.
Uncommitted Space: This space is used on demand basis
```

Figure 03 Details of Parameters

The details of the JVM parameters are as below.

2.1) Maximum and Minimum heap size -Xmx2048m -Xms2048m

2.2) Configures a large Java heap to take advantage of the large memory system and 64 bit processors, size of memory allocation in JVM is increased in 64 bit systems

2.3) Maximum size of New Generation in mega bytes -XX:MaxNewSize=64m & -XX:NewSize=64m

2.4) The parameters `NewSize` and `MaxNewSize` bound the *young* generation size from below and above. Setting these equal to one another fixes the *young* generation, just as setting `-Xms` and `-Xmx` equal fixes the total heap size. This is useful for tuning the *young* generation at a finer granularity than the integral multiples allowed by `NewRatio`.

2.5) For most applications in our Jboss EAP 4.3 platform the *permanent* generation is not relevant to garbage collector performance. However, we did aboserve some applications dynamically generate and load many classes. For instance, some implementations of JSP™ pages do this. Taking this factor into consideration, the maximum *permanent* generation size was increased to -XX:MaxPermSize=128m.

2.6) This size is double to the default permgen setup on 64 bit env.

2.7) The parameter `SurvivorRatio` can be used to tune the size of the survivor spaces If survivor spaces are too small, copying collection overflows directly into the *tenured* generation. If survivor spaces are too large, they will be uselessly empty. At each garbage collection the virtual machine chooses a threshold number of times an object can be copied before it is tenured. This threshold is chosen to keep the survivors half full -XX:SurvivorRatio=8

2.8) Desired percentage of survivor space used after scavenge, In our test we Allowed 90% of the survivor spaces to be occupied instead of the default 50%, allowing better utilization of the survivor space memory. -XX:TargetSurvivorRatio=90

2.9) The MaxTenurubgThreshold, Allows short-lived objects a longer time period to die in the young generation (and hence, avoid promotion). A consequence of this setting is that minor GC times can increase due to additional objects to copy. This value and survivor space sizes may need to be adjusted so as to balance overheads of copying between survivor spaces versus tenuring objects that are going to live for a long time, we left this setting to default which is -XX:MaxTenuringThreshold=0

2.10) CMSInitiatingOccupancyFraction: With the serial collector a major collection is started when the tenured generation becomes full and all application threads are stopped while the collection is done. In contrast a concurrent collection should be started at a time such that the collection can finish before the tenured generation becomes full. There are several ways a concurrent collection can be started.

2.11) The concurrent collector keeps statistics on the time remaining before the tenured generation is full (T-until-full) and on the time needed to do a concurrent collection (T-collect). When the T-until-full approaches T-collect, a concurrent collection is started. This test is appropriately padded so as to start a collection conservatively early.

2.12) A concurrent collection will also start if the occupancy of the tenured generation grows above the initiating occupancy (i.e., the percentage of the current heap that is used before a concurrent collection is started). The initiating occupancy by default is set to about 68%. It can be set with the parameter CMSInitiatingOccupancyFraction which can be set on the command line with the flag

2.13) -XX:CMSInitiatingOccupancyFraction=<nn>
The value <nn> is a percentage of the current tenured generation size. We have reduced the percentage from 68% to 60% as we were able to get the best performance on this.

2.14) LargePageSizeInBytes : We got the best efficiency out of the memory management system of our server. However we have just allocated 5MB to of space to this, but it can be increased based on the applications need.
Note that with larger page sizes we can make better use of virtual memory hardware resources
-XX:LargePageSizeInBytes=5m

2.15) A throughput collector is a generational collector similar to the serial collector but with multiple threads used to do the minor collection. The major collections are essentially the same as with the serial collector. By default on a host with *N* CPUs, the throughput collector uses *N* garbage collector threads in the minor collection. The number of garbage collector threads can be controlled with a command line option (see below). On a host with 1 CPU the throughput collector will likely not perform as well as the serial collector because of the additional overhead for the parallel execution (e.g., synchronization costs). On a host with 2 CPUs the throughput collector generally performs as well as the serial garbage collector and a reduction in the minor garbage collector pause times can be expected on hosts with more than 2 CPUs.

2.16) throughput collector can be enabled by using command line flag `-XX:+UseParallelGC`. The number of garbage collector threads can be controlled with the `ParallelGCThreads` command line option (`-XX:ParallelGCThreads=<desired number>`). If explicit tuning of the heap is being done with command line flags the size of the heap needed for good performance with the throughput collector is to first order the same as needed with the serial collector. Turning on the throughput collector should just make the minor collection pauses shorter. Because there are multiple garbage collector threads participating in the minor collection there is a small possibility of fragmentation due to promotions from the *young* generation to the *tenured* generation during the collection. Each garbage collection thread reserves a part of the *tenured* generation for promotions and the division of the available space into these "promotion buffers" can cause a fragmentation effect.
We used -XX:ParallelGCThreads=10, as we wanted not to use too much of CPUs threads neither we wanted to use default because of the larger size of heap and due to more then one CPU used.

Note: This number should be changed based on the CPU count, heap size and other parameters of JVM settings.

2.17) Stopping the explicit garbage collection: Applications can interact with garbage collection is by invoking full garbage collections explicitly, such as through the `System.gc()` call. These calls force major collection, and inhibit scalability on large systems. The performance impact of explicit garbage collections can be measured by disabling explicit garbage collections using the flag `-XX:+DisableExplicitGC`.

2.18) This also makes our system more robust as we can control the GC of the system based on the env needs.

2.19) Concurrent Mark Sweep collector: This collector may deliver better response time properties for the application (i.e., low application pause time). It is a parallel and mostly-concurrent collector and and can be a good match for the threading ability of an large multi-processor systems. -XX:+UseConcMarkSweepGC -XX:+UseParNewGC

2.20) ThreadStackSize : Is used based on the env and the underlying platform, It can be different for different systems.

2.21) We used 1024 as is best suited for the AMD based 64 bit processors and can be used for large scale systems

2.22) -XX:ThreadStackSize=1024

2.23) Distributed garbage collection: One of the most commonly encountered uses of explicit garbage collection occurs with RMI's distributed garbage collection (DGC). Applications using RMI refer to objects in other virtual machines. Garbage can't be collected in these distributed applications without occasional local collection, so RMI forces periodic full collection. Which is a type of Stop the world collection, this can cause a significant overhead to the application performance.

2.24) The default value is 1 DGC per min, In our test scenarios we have reduced the DGC cycle to 1 per hr, this gives us an additional boost of performance during the endurance test and during the peak/stress test.

2.25) -Dsun.rmi.dgc.client.gcInterval=3600000

2.26) -Dsun.rmi.dgc.server.gcInterval=3600000

V. Monitoring & Profiling

Discussing monitoring (extracting high level statistics from a running application) or profiling (incrementing an application to provide detailed performance statistics) are subjects, which are worthy of White Papers in their own right.

In our test we used standard JVM monitoring tools, which give us a good and necessary amount of data.

VI. Coding Practice

This section will cover coding level changes that you can make which will make an impact on performance. For the purpose of this initial draft of the Java Tuning White Paper examples of the kinds of coding level changes that can have an impact on performance are taking advantage of new language features like NIO and the Concurrency utilities.

The New I/O API's (or NIO) offer improved performance for operations like memory-mapped files and scalable network operations. By using NIO developers may be able to significantly improve performance of memory or network intensive applications.

new Java language features that impact performance is the set of Concurrency Utilities. Increasingly server applications are going to be targeting platforms with multiple CPU's and multiple cores per CPU. In order to best take advantage of these systems applications must be designed with multi-threading in mind. Classical multi-threaded programming is very complex and error prone due to subtleties in threads interactions such as race conditions. Now with the Concurrency Utilities developers finally have a solid set of building blocks upon which to build scalable multi-threaded applications while avoiding much of the complexity of writing a multi-threaded framework.

3) Jboss[[16]]/Apache Sliming[9][10]

3.1) Reducing the thread count for tomcat access: Tomcat is not connected directly by the application, so why we need to open so many connections to tomcat engine, the hash table which needs to be rehashed due to the increase in connection need to be kept to very low we will keep it to 10.

3.2) Disabling deployment parameter: As a default this parameter is on and the server scans for the new class every 5 seconds which is counterproductive, as we don't use hot deployment in our Jboss EAP 4.3 application farms.

3.3) Remote Method Invocation (RMI): 01>by default, JBoss creates a new thread for every RMI request that comes in. This is not generally efficient on a large system 02>Secondly, it can be dangerous to allow unrestrained connections in the case of performance or traffic spikes or run-away connection creating clients. To remedy this you should consider switching to the pooled invoker.

3.4) Logging: -Logging has a profound effect on performance. Changing the logging level to TRACE can bring the JBossAS to a crawl. Changing it to ERROR (or WARN) can speed things up dramatically. By default, JBoss logs both to the console and server.log and by default it uses level "INFO".Consider not logging to System.out (you may still want to redirect it to catch JVM errors)

3.5) Consider changing the log level to ERROR. Remember that JBoss watches its log4j config file for changes and you can always change configuration at runtime. Add a category filter for your Java class hierarchy.

3.6) We have used Info and do suggest to use Info in Staging and Production env, in addition to this we turned off logging verbosity and turned off console logging in production .

3.7) Mail Services: In our case most of the applications are not using JAF (Java Activation Framework), hence these class are not required in the container.

3.8) Jboss Hypersonic, Jboss MQ, JMS: We are not using any of such properties in our env, hence its an overhead for the system and we removed it from our shared env. These additional packages will be deployed on demand basis.

3.9) HTTPInvoker (which lets you tunnel RMI over HTTP): This function we are not using and it must not be used as it allows you to tunnel through HTTP protocol

3.10) Jboss Scheduler: We are not using any scheduling within the application layer and within the Jboss container, It takes extra overhead to run the scheduler threads and must be avoided to get the best performance

3.11) Vendor Specific SQL exception: We must disable it as we are not using any db vendor specific APIS.

3.12) JBoss UUID key generation: It should be done on demand

3.13) Tomcat via HTTP: It must never be allowed as it's a breach of security, the communication to tomcat must always go through modjk

3.14) Disable hot deployment in production

3.15) Clustering must be done only for dedicated env.

3.16) Apache complication will be use the core module (Appendix 04) only , addition of modules will be done on demand and with the justification

3.17) Farm deployment /replicated deployment must be avoided

3.18) XA or Two Phase commits transaction: Don't use XA versions unless you really know you need them. XA connections do not have good performance.

3.19) Use database specific "ping" support where available for "check-connection" or use database-specific driver fail-over support rather than checking connections at all. (Remember that not all tuning options may be feasible in your environment, we're talking optimal).

4) Storage tunning :

We at NBC Universal were using mount points, which was containing application code, log and web content. This practice seemed good during the ear of physical server, as there was limitation of local space on the box.

Where in with VM we have this flexibility to have a bigger /opt and it has a dedicated communication between the VM/ESX server to the storage.

We found approximately 200% increase in performance when we moved to VM local Figure 04.

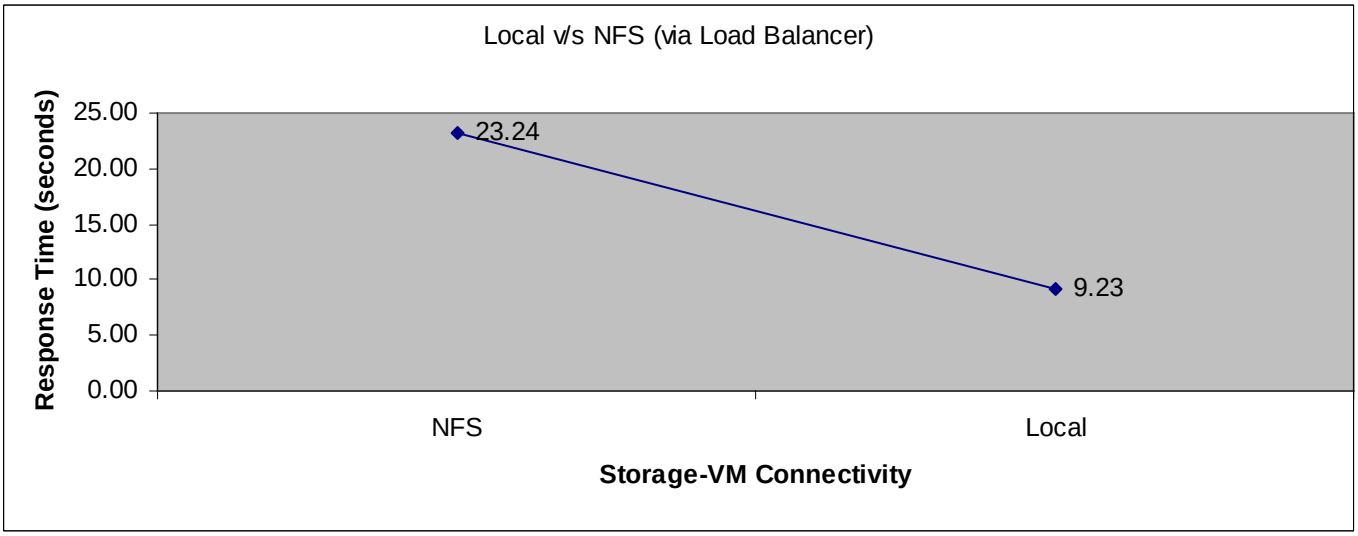


Figure 04

In addition to the above we also tested the scenario in which the call is not routes through the load balancer while communicating from App/Web VM to storage, for this we took help of Jan Kaliki and Monsanto, Joseph(VM architecture team). With their help we were able to design a virtual NIC, which can help us rout the traffic without routing it through the F5 load balance, figure 05.

We did see a significant improvement over the baseline, but comparing it with local still it was not able to match the performance.

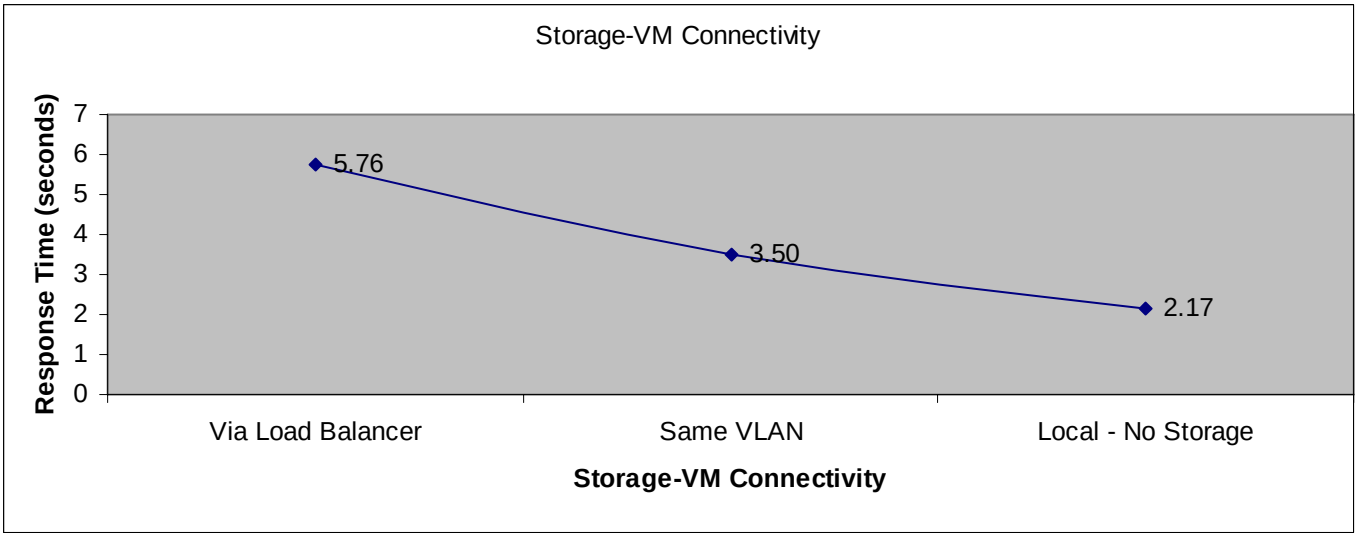


Figure 05

-

Appendix 01

Case#	Params	Description
Case 1	-Xmx1024m -Xms1024m -XX:MaxNewSize=128m -XX:NewSize=128m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=64m -XX:ParallelGCThreads=20 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024	3 GB RAM , 2 VCPUS with no blocking of DGC(distributed garbage collection)
Case 2	-Xmx1124m -Xms1124m -XX:MaxNewSize=64m -XX:NewSize=64m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=30m -XX:ParallelGCThreads=20 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	3 GB RAM , 2 VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr)

-

Case 3	-Xmx1303m -Xms1303m -XX:MaxNewSize=64m -XX:NewSize=64m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=50 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=10m -XX:ParallelGCThreads=30 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	3 GB RAM, 2 VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr)
Case 4	-Xmx1500m -Xms1500m -XX:MaxNewSize=128m -XX:NewSize=128m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=10m -XX:ParallelGCThreads=30 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	4 GB RAM, 4VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr)

-

Case 5	-Xmx1700m -Xms1700m -XX:MaxNewSize=128m -XX:NewSize=128m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=10m -XX:ParallelGCThreads=20 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	4 GB RAM, 4VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr)
Case 6	-Xmx2048m -Xms2048m -XX:MaxNewSize=256m -XX:NewSize=256m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=5m -XX:ParallelGCThreads=20 ~~-XX:-DisableExplicitGC~~ ~~-XX:-RelaxAccessControlCheck~~ -XX:+StringCache -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	4 GB RAM, 4VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr)

-

Case 7	-Xmx1900m -Xms1900m -XX:MaxNewSize=128m -XX:NewSize=128m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=5m -XX:ParallelGCThreads=20 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	4 GB RAM, 4VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr)
Case 8	-Xmx2048m -Xms2048m -XX:MaxNewSize=64m -XX:NewSize=64m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=5m -XX:ParallelGCThreads=10 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	4 GB RAM, 4VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr)

-

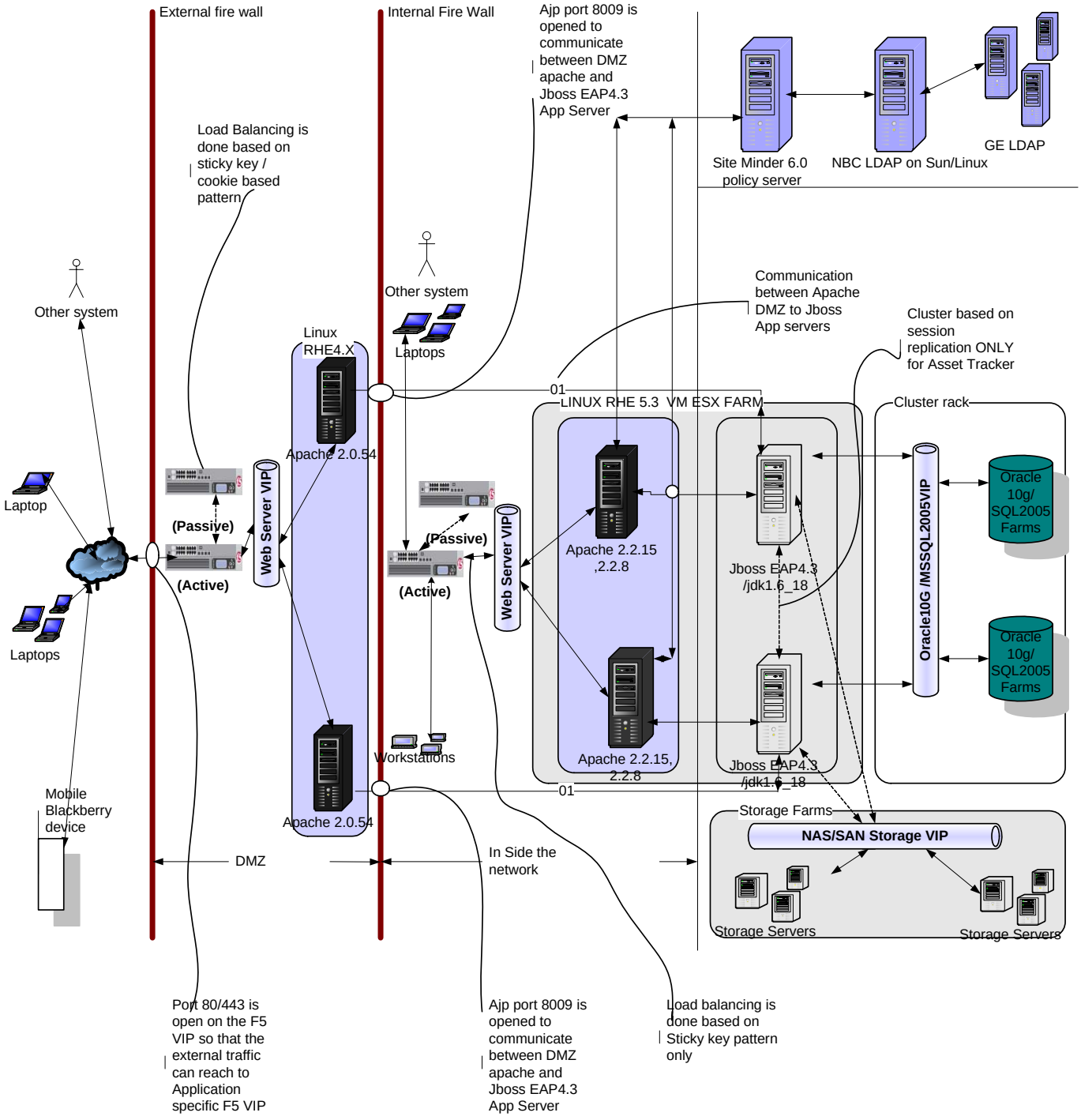
Case 9	-Xmx2048m -Xms2048m -XX:MaxNewSize=64m -XX:NewSize=64m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=5m -XX:ParallelGCThreads=10 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	4 GB RAM, 2VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr)
Case 10	-Xmx2048m -Xms2048m -Xss512k -XX:MaxNewSize=128m -XX:NewSize=128m -XX:MaxPermSize=128m -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 -XX:LargePageSizeInBytes=5m -XX:ParallelGCThreads=10 -XX:-DisableExplicitGC ~~-XX:-RelaxAccessControlCheck~~ ~~-XX:+StringCache~~ -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:ThreadStackSize=1024 -Dsun.rmi.dgc.client.gcInterval=3600000	4 GB RAM, 2VCPUS --> increasing max and min reducing other and adding DGC(distributed Garbage collection every 01 hr) with reducing the default Stack Size

Appendix 02

Project used for testing these performance parameters:

Project name	Project Group	Type of test	Remarks
Asset Tracker	IT TV Operations Mgmt Systems	JVM, JDK, NOISE, Endurance, Apache and Jboss trimming	This is a unique app as its horizontally clustered and shares the session between nodes
IRIS	IT Film International (Legal)	JVM, JDK, NOISE , Endurance, Apache and Jboss trimming	This app we wanted to improve the performance without touching the code.
NBC Sports	IT TV Operations Mgmt Systems	Noise, Endurance and Storage	For this app we wanted to resize the env
Facilities Apps	IT TV Operations Mgmt Systems	Noise, Endurance and Storage	We wanted to see a deviation when there are many small app and how do they react to the changes in the env
NOTE:	In essence we too different type of applications which uses the env in different way just to make sure our results are more and more close to the actual production env.		

Appendix03 Architecture used




```
OS Details:Red Hat Enterprise Linux Server release 5.3 Tikanga)
Kernel:2.6.18-128.7.1.el5 x86_64 x86_64 x86_64 GNU/Linux
Java/JDK
java version"1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b18)
Java HotSpot(TM) 64-Bit Server VM
Web Server:- Apache2.2.8, 2.2.15
App Server:- Enterprise Application Platform Jboss 4.3
```

UNIT	NBCUN Web Architecture
Project	J2EE/JBOSS EAP 4.3.4 Non DMZ/DMZ
Group	Media
Developed by	Ruchir C, Alpesh Vaghela
Version	1.0.0.2

Appendix 04

Apache version	Modules use in old setup	Modules used in new setup	Remarks
2.2.8/2.2.15	core_module (static) authn_file_module (static) authn_default_module (static) authz_host_module (static) authz_groupfile_module (static) authz_user_module (static) authz_default_module (static) auth_basic_module (static) include_module (static) filter_module (static) log_config_module (static) env_module (static) setenvif_module (static) mpm_prefork_module (static) http_module (static) mime_module (static) status_module (static) autoindex_module (static) asis_module (static) negotiation_module (static) dir_module (static) actions_module (static) userdir_module (static) alias_module (static) so_module (static) sm_module (shared) authn_dbm_module (shared) authn_anon_module (shared) authn_alias_module (shared) authz_dbm_module (shared) authz_owner_module (shared) auth_digest_module (shared) file_cache_module (shared) cache_module (shared) disk_cache_module (shared) mem_cache_module (shared) ext_filter_module (shared) deflate_module (shared) logio_module (shared) mime_magic_module (shared) expires_module (shared) headers_module (shared) usertrack_module (shared) proxy_module (shared) proxy_connect_module (shared) proxy_ftp_module (shared) proxy_http_module (shared)	core_module (static) authn_file_module (static) authn_default_module (static) authz_host_module (static) authz_groupfile_module (static) authz_user_module (static) authz_default_module (static) auth_basic_module (static) include_module (static) filter_module (static) log_config_module (static) env_module (static) setenvif_module (static) version_module (static) mpm_prefork_module (static) http_module (static) mime_module (static) status_module (static) autoindex_module (static) asis_module (static) cgi_module (static) negotiation_module (static) dir_module (static) actions_module (static) userdir_module (static) alias_module (static) so_module (static) sm_module (shared) info_module (shared) vhost_alias_module (shared) rewrite_module (shared) jk_module (shared)	

	proxy_ajp_module (shared) proxy_balancer_module (shared) dav_module (shared) info_module (shared) suexec_module (shared) cgi_module (shared) dav_fs_module (shared) vhost_alias_module (shared) speling_module (shared) rewrite_module (shared) jk_module (shared)		
Note : Please refer to apache [9][10] for modules details			

VII. CONCLUSIONS

This white paper presents various details by which the overall performance of the system can be increased without touching the code. It also give a technical insight that how JVM settings, change in storage, change in compiling different modules in apache, trimming Jboss parameters and trying to start Jboss with different startups may impact the system.
This document can be used as a living doc for reference in architecting a highly scalable, avilable distributed evn.

VIII. ACKNOWLEDGMENT

I would like to acknowledge the contributions of the following groups and individuals who helped us during the project and to develop this white paper.

Alpesh Vaghela –Web Architecture

For his continuous work in getting details of the setup and figuring out various parameters, which can help us, improve the performance

Vaidyanathan Kothandaraman- Web Architecture

For his guidance, motivation , persisted to make the env better and better and his strive for excellence.

Agasthi Kothurkar –Web Architecture

For setting up various env within a short notice and working tirelessly with the performance team to get various performances test done

Anthony Sarabia- Web Operations

For always motivating me to do things which is never done in NBC Universal

Maskara Arvind –Web Operations

For supporting my team, giving Ideas to improve.

Jan Kalicki, -Virtualization COE

For architecting the new virtual NIC so that the application server can communicate directly to the storage without routing through the F5 load balancer

Joseph Monsanto –VM Architect

For setting up the virtual NIC on the VM server

Joanne Marshall,- Sr Sever Engineer

For providing continuous support in changing various OS parameters and helping us complete this endeavour.

IX. REFERENCES

[1] http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp#BehavioralOptions
[2] http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html
[3] http://java.sun.com/docs/hotspot/gc1.4.2/example.html
[4] http://www.ibm.com/developerworks/java/library/j-jtp11253/
[5] http://www.ibm.com/developerworks/ibm/library/i-garbage1/
[6]http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/tprf_tunejvm.html
[7] http://java.sun.com/docs/hotspot/gc1.4.2/faq.html
[8] http://java.sun.com/developer/technicalArticles/Programming/turbo/
[9] http://httpd.apache.org/docs/2.2/
[10] http://httpd.apache.org/docs/2.2/mod/
[11] http://www.javaperformancetuning.com/
[12] http://www.artima.com/index.jsp
[13]Tim Lindholm and Frank Yellin ***The Java™ Virtual Machine Specification, Second Edition***
[14] Joshua Engel Programming for the Java(TM) Virtual Machine
[15] http://docs.jboss.com/jbossas/guides/clusteringguide/r2/en/html_single/#cluster.chapt
[16] http://community.jboss.org/wiki/JBossASTuningSliming

-