**Common Performance problem:**

* Java engine hangs during startup.
* Java engine takes a long time to return a response. For example, search operations take long time to respond in portal.
* Time-out error for portal and other java applications.
* Long running or frequent garbage collection activity (Full GCs).
* Heavy paging activity.
* High CPU usage.
* Connectivity between servernode and message server fails (missed broadcast, delay while getting response from message server/servernode. All -33X exit codes)
* System or applications threads maxed out.
* System slows down and crashes with out of memory (exitcode 666). <std\_serverN.out> contains java.lang.OutOfMemoryError or Out of memory. System will write heap dumps if the parameters as per Sap note [1004255](https://websmp230.sap-ag.de/sap(bD1lbiZjPTAwMQ==)/bc/bsp/spn/sapnotes/index2.htm?numm=1004255) are maintained.
* Unable to login to java applications or login takes a lot of time.
* Intermittent servernode restarts.
* Portal or other java application pages remain blank after login.

**Commonly seen reasons behind performance issues**

* Insufficient heap, permanent or physical memory.
* Insufficient number of application and/or system threads
* Too many logs are written due to log severity settings.
* Message server connectivity issues - problem with message server timeout settings.
* Java threads being blocked by certain applications.
* Slow response from LDAP server.
* Incorrect JVM parameters settings.
* Outdated JVM used.
* High CPU consumption by Java processes.
* Outdated database statistics lead to slow portal performance.
* Network issues.
* Certain applications consuming majority of heap.
* Inconsistent support pack levels leading system to crash.
* Timed out java services.
* Dispatcher node hangs (valid for releases lower that Netweaver 7.1).
* Communication error between java and SCS instances..
* All system threads are in use.
* All application threads are in use.

# Top 10 Most Common Java Performance Problems

Posted by: [Theodora Fragkouli](http://www.javacodegeeks.com/author/theodora-fragkouli/) in [Enterprise Java](http://www.javacodegeeks.com/category/java/enterprise-java/) February 9th, 2015

Java performance is an issue of interest for all Java application developers, since making an application fast is as important as making it functional. Steven Haines uses his personal experience on Java performance issues to conclude that [most of them have common root causes](http://info.appdynamics.com/Top10JavaPerformanceProblems_eBook.html?utm_source=javacodegeeks&utm_medium=sponsorship&utm_campaign=sponsored%20post&utm_content=top%2010%20java%20performance%20problems-ebook&utm_term=java). So, as a performance analyst, Haines sorts the basic performance issues to three basic categories:

**Database problems**, that mostly have to do with persistence configuration, caching or database connection thread pool configuration.

**Memory problems**, that usually are garbage collection misconfiguration or memory leaks.

**Concurrency problems**, and basically deadlocks, gridlocks and thread pool configuration problems.

Let’s delve into each category…

Database

Since database is the basic component of an application functionality, it also is the basic root of performance issues. Problems may occur due to wrong use of access to the database, bad connection pool size or missing tuning.

Persistence configuration

Even though today Hibernate and other JPA implementations provide fine tuning of database access, there are some more options such as eager or lazy fetching, that may lead to long response times and database overheads. Eager fetching make less but more complex database calls, whereas lazy fetching makes more but more simple and fast database calls.

Problems occur when the load of the application increases and it causes a much bigger database load. So, in order to fix this, you can take a look at the business transaction counters, the database counters, but basically at the correlation between a business transaction and database calls. [To avoid such problems](http://info.appdynamics.com/Top10JavaPerformanceProblems_eBook.html?utm_source=javacodegeeks&utm_medium=sponsorship&utm_campaign=sponsored%20post&utm_content=top%2010%20java%20performance%20problems-ebook&utm_term=java) you must understand well the persistence technology used, set correctly all configuration options, so as to pair their functionality with your business domain needs.

Caching

Caching has optimized the performance of applications, since in-memory data is faster to access than persisted ones. Problems are caused when no caching is used, so every time a resource is needed it is retrieved from database. When caching is used, problems occur due to its bad configuration. Basic things to notice here are the fixed size of a cache and the distributed cache configuration. Cached objects are stateful, unlike pools that provide stateless objects. So a cache must be properly configured so as not to exhaust memory. But what if a removed object is requested again? This ‘miss’ ratio must be configured in cache settings, along with the memory.

Distributed caching may also cause problems. Synchronization is necessary when caches are set to multiple servers. Thus, a cache update is propagated to caches in all servers. This is how consistency is achieved, but it is a very expensive procedure. When caching is used correctly the application load increase does not increase the database load, but when the caching settings are wrong, then the database load increases, causing CPU overhead an even disk I/O rate.

In order to troubleshoot this problem you should first examine the database performance so as to decide if cache is needed or not. Then, you should determine the cache size, using the hit ratio and miss ratio metrics. You can avoid facing caching problems though, by planning correctly your application before building it. Make sure to use serialization and techniques that provide a scalable application.

Pool connections

Pool connections are usually created before starting the application, since they are expensive to create. A pool of connections is shared across the transactions and the pool size limits the database load.

Pool size is important. Not enough connections make business transactions to wait and the database is under-utilized. On the other hand, too many connections cause bigger response time and database overload. In order to solve this problem you must check whether your application is waiting for a new connection or for a database query to be executed. You can always avoid it though, by optimising the database, test the application with different pool size to check which one fits the case.

Memory

Memory problems have to do with Garbage Collector and memory leaks.

Garbage Collector

Garbage collection may cause all threads to stop in order to reclaim memory. When this procedure takes too much time or occurs too often, then there is a problem. Its basic symptoms are the CPU spikes and big response times. [To solve this](http://info.appdynamics.com/Top10JavaPerformanceProblems_eBook.html?utm_source=javacodegeeks&utm_medium=sponsorship&utm_campaign=sponsored%20post&utm_content=top%2010%20java%20performance%20problems-ebook&utm_term=java) you can configure your -verbosegc params, use a performance monitoring tool to find major GC occurs, and a tool to monitor heap usage and possible CPU spikes. It is almost impossible to avoid this problem, though can limit it by configuring heap size and cycling your JVM.

Memory leaks

Memory leaks in Java may occur in different ways than C or C++, since they are more of a reference management issue. In Java a reference to an object may be maintained even though it may not be used again. This may lead to an OutOfMemory error and demand a JVM restart. When the memory usage is increased and the heap runs out of memory then the memory leak issue has occurred. To solve it, you could configure the JVM params properly. To avoid having to deal with memory leaks, you can pay attention while coding to memory leak – sensitive Java collections, or session management. You can share memory leaks avoid tips with colleagues, have an expert take a look at your application code, and use tools to avoid memory leaks and analyze heap.

Concurrency

Concurrency occurs when several computations are executed at the same time. Java uses synchronization and locks to manage multithreading. But synchronization can cause thread deadlocks, gridlocks and thread pool size issues.

Thread deadlocks

Thread deadlocks occur when two or more threads are trying to access same resources and the one is waiting for the other one to release a resource and vice versa. When a deadlock occurs the JVM exhausts all threads and the application is getting slower. Deadlocks are very difficult to reproduce. So, a way to solve a deadlock problem is to capture a thread dump while two threads are deadlocked and examine stack traces of the threads. To avoid this problem you’d better make your application and its resources as immutable as possible, make use of synchronization and check for potential threads interactions.

Thread gridlocks

Thread gridlocks may occur when too much synchronization is used and thus too much time is spent waiting for a single resource. To notice this, you must have both slow response times and low CPU utilization, since many threads try to access the same code part and they are waiting for the one that has it to finish. So, how can you solve this? You must first check where your threads are waiting and why. Then, you should eliminate the synchronization requirements according to your business requirements.

Thread pool configuration locks

When an application uses an application server or a web container, a thread pool is used to control the concurrently processed requests. If this thread pool is small, then the requests will wait a lot, but if it is too large, then the processing resources will be too busy. So, at a small pool size the CPU is underutilized but the thread pool utilization is 100%, whereas at a large pool size the CPU is very busy.

You can [troubleshoot this problem easily](http://info.appdynamics.com/Top10JavaPerformanceProblems_eBook.html?utm_source=javacodegeeks&utm_medium=sponsorship&utm_campaign=sponsored%20post&utm_content=top%2010%20java%20performance%20problems-ebook&utm_term=java), by checking your thread pool utilization and CPU utilization and decide whether to increase or decrease the pool size. To avoid it, you must tune the thread pool, and that is not so easy to do.

Finally, two basic issues that may occur are the performance issue to be an afterthought, or the performance issue to be noticed by the end users.

The first case is a common problem. Usually developers create an application that is functional but fails in performance tests. To solve this they usually have to make an architectural review of the application, where performance analysis tools seem very handy. To avoid this problem, try to test performance while developing the application, so continuous integration is the key.

[](http://info.appdynamics.com/Top10JavaPerformanceProblems_eBook.html?utm_source=javacodegeeks&utm_medium=sponsorship&utm_campaign=sponsored%20post&utm_content=top%2010%20java%20performance%20problems-ebook&utm_term=java)For the second case, what happens when end users of the application inform you that there are performance issues? There are tools to avoid this case, such as JMX to check your servers behavior. Business Transaction Performance results combined with JMX results may help too. Method-level response time checks all methods called in a business transaction and finds hotspots of the application. So, you’d better make use of one of these tools, so that end users will never alert you for performance.

Performance problems are one of the biggest challenges to expect when designing and implementing Java EE related technologies. Some of these common problems can be faced when implementing either lightweight or large IT environments; which typically include several distributed systems from Web portals & ordering applications to enterprise service bus (ESB), data warehouse and legacy Mainframe storage systems.

It is very important for IT architects and Java EE developers to understand their client environments and ensure that the proposed solutions will not only meet their growing business needs but also ensure a long term scalable & reliable production IT environment; and at the lowest cost possible. Performance problems can disrupt your client business which can result in short & long term loss of revenue.

This article will consolidate and share the top 10 causes of Java EE performance problems I have encountered working with IT & Telecom clients over the last 10 years along with high level recommendations.

Please note that this article is in-depth but I'm confident that this substantial read will be worth your time.

**#1 - Lack of proper capacity planning**

I'm confident that many of you can identify episodes of performance problems following Java EE project deployments. Some of these performance problems could have a very specific and technical explanation but are often symptoms of gaps in the current capacity planning of the production environment.

Capacity planning can be defined as a comprehensive and evolutive process measuring and predicting current and future required IT environment capacity. A proper implemented capacity planning process will not only ensure and keep track of current IT production capacity and stability but also ensure that new projects can be deployed with minimal risk in the existing production environment. Such exercise can also conclude that extra capacity (hardware, middleware, JVM, tuning, etc.) is required prior to project deployment.

In my experience, this is often the most common "process" problem that can lead to short- and long- term performance problems. The following are some examples.

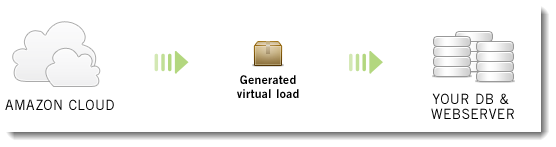
|  |  |
| --- | --- |
| **Problems observed** | **Possible capacity planning gaps** |
| A newly deployed application triggers an overload to the current Java Heap or Native Heap space (e.g., java.lang.OutOfMemoryError is observed). | -Lack of understanding of the current JVM Java Heap (YoungGen and OldGen spaces) utilization  -Lack of memory static and / or dynamic footprint calculation of the newly deployed application  -Lack of performance and load testing preventing detection of problems such as Java Heap memory leak |
| A newly deployed application triggers a significant increase of CPU utilization and performance degradation of the Java EE middleware JVM processes. | -Lack of understanding of the current CPU utilization (e.g., established baseline)  -Lack of understanding of the current JVM garbage collection healthy (new application / extra load can trigger increased GC and CPU)  -Lack of load and performance testing failing to predict the impact on existing CPU utilization |
| A new Java EE middleware system is deployed to production but unable to handle the anticipated volume. | -Missing or non-adequate performance and load testing performed  -Data and test cases used in performance and load testing not reflecting the real world traffic and business processes  -Not enough bandwidth (or pages are much bigger than capacity planning anticipated) |

One key aspect of capacity planning is load and performance testing that everybody should be familiar with. This involves generating load against a production-like environment or the production environment itself in order to:

* Determine how much concurrent users / orders volumes your application(s) can support
* Expose your platform and Java EE application bottlenecks, allowing you to take corrective actions (middleware tuning, code change, infrastructure and capacity improvement, etc.)

There are several technologies out there allowing you to achieve these goals. Some load-testing products allow you to generate load from inside your network from a test lab while other emerging technologies allow you to generate load from the "Cloud".

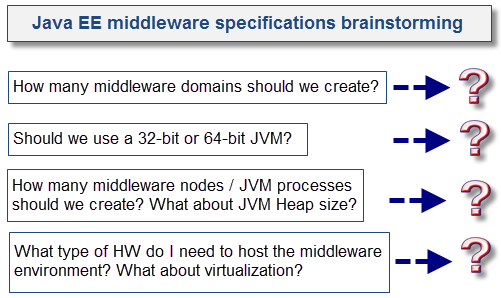
I'm currently exploring the free version of [Load Tester](http://txt.couchware.com/medias/jump?hid=2595&cid=447&mid=789), a new load testing tool I found allowing you to record test cases and generate load from inside your network or [from the Cloud](http://txt.couchware.com/medias/jump?hid=2595&cid=447&mid=789).



Regardless of the load and performance testing tool that you decide to use, this exercise should be done on a regular basis for any dynamic Java EE environments and as part of a comprehensive and adaptive capacity planning process. When done properly, capacity planning will help increase the service availability of your client IT environment.

**#2 - Inadequate Java EE middleware environment specifications**

The second most common cause of performance problems I have observed for Java EE enterprise systems is an inadequate Java EE middleware environment and / or infrastructure. Not making proper decisions at the beginning of new platform can result in major stability problems and increased costs for your client in the long term. For that reason, it is important to spend enough time brainstorming on required Java EE middleware specifications. This exercise should be combined with an initial capacity planning iteration since the business processes, expected traffic, and application(s) footprint will ultimately dictate the initial IT environment capacity requirements.



Now, find below typical examples of problems I have observed in my past experience:

* Deployment of too many Java EE applications in a single 32-bit JVM
* Deployment of too many Java EE applications in a single middleware domain
* Lack of proper vertical scaling and under-utilized hardware (e.g., traffic driven by one or just a few JVM processes)
* Excessive vertical scaling and over-utilized hardware (e.g., too many JVM processes vs. available CPU cores and RAM)
* Lack of environment redundancy and fail-over capabilities

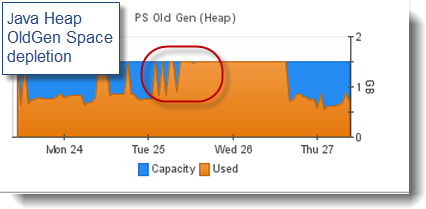
Trying to leverage a single middleware and / or JVM for many large Java EE applications can be quite attractive from a cost perspective. However, this can result in an operation nightmare and severe performance problems such as excessive JVM garbage collection and many domino effect scenarios (e.g., Stuck Threads) causing high business impact (e.g., App A causing App B, App C, and App D to go down because a full JVM restart is often required to resolve problems).

Recommendations

* Project team should spend enough time creating a proper operation model for the Java EE production environment.
* Attempt to find a good "balance" for your Java EE middleware specifications to provide to the business & operation team proper flexibility in the event of outages scenarios.
* Avoid deployment of too many Java EE applications in a single 32-bit JVM. The middleware is designed to handle many applications, but your JVM may suffer the most.
* Choose a 64-bit over a 32-bit JVM when it is required but combine with proper capacity planning and performance testing to ensure your hardware will support it.

**#3 - Excessive Java VM garbage collections**

Now let's jump to pure technical problems starting with excessive JVM garbage collection. Most of you are familiar with this famous (or infamous) Java error: java.lang.OutOfMemoryError. This is the result of JVM memory space depletion (Java Heap, Native Heap, etc.).



I'm sure middleware vendors such as Oracle and IBM could provide you with dozens and dozens of support cases involving JVM OutOfMemoryError problems on a regular basis, so no surprise that it made the #3 spot in our list.

Keep in mind that a garbage collection problem will not necessarily manifest itself as an OOM condition. Excessive garbage collection can be defined as an excessive number of minor and / or major collections performed by the JVM GC Threads (collectors) in a short amount of time leading to high JVM pause time and performance degradation. There are many possible causes:

* Java Heap size chosen is too small vs.JVM concurrent load and application(s) memory footprint.
* Inappropriate JVM GC policy used.
* Your application(s) static and / or dynamic memory footprint is too big to fit in a 32-bit JVM.
* The JVM OldGen space is leaking over time \* quite common problem \*; excessive GC (major collections) is observed after few hours / days.
* The JVM PermGen space (HotSpot VM only) or Native Heap is leaking over time \* quite common problem \*; OOM errors are often observed over time following application dynamic redeployments.
* Ratio of YoungGen / OldGen space is not optimal to your application(s) (e.g., a bigger YoungGen Space is required for applications generating massive amount of short lived objects). A bigger OldGen space is required for applications creating lot of long lived / cached Objects.
* The Java Heap size used for a 32-bit VM is too big leaving small room for the Native Heap. Problems can manifest as OOM when trying to a new Java EE application, creating new Java Threads or any computing task that requires native memory allocations.

Before pointing a finger at the JVM, keep in mind that the actual "root" cause can be related to our #1 & #2 causes. An overloaded middleware environment will generate many symptoms, including excessive JVM garbage collection.

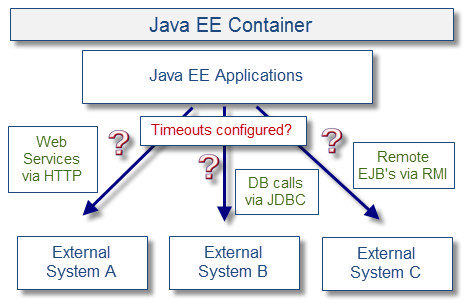
Proper analysis of your JVM related data (memory spaces, GC frequency, CPU correlation, etc.) will allow you to determine if you are facing a problem or not. Deeper level of analysis to understand your application memory footprint will require you to analyze JVM Heap Dumps and / or profile your application using profiler tools (*such as JProfiler)* of your choice.

Recommendation

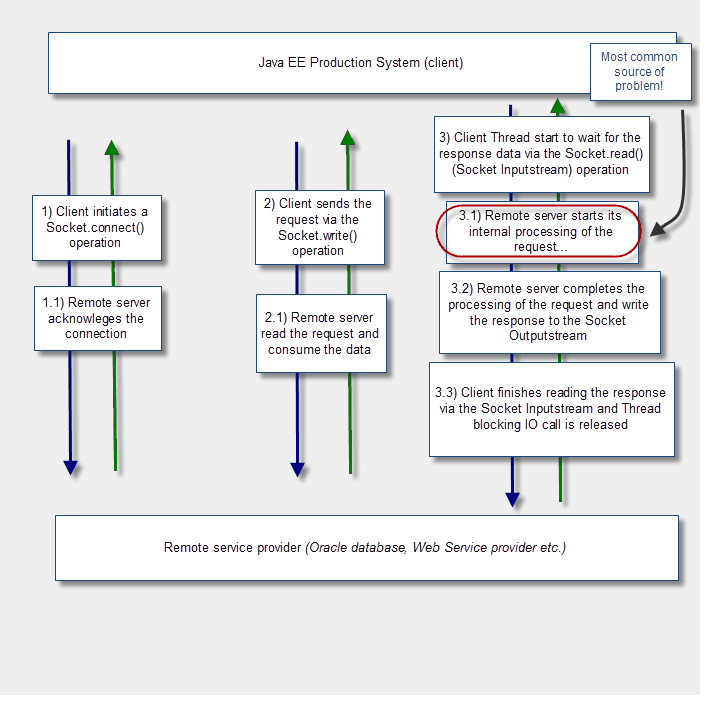
* Ensure that you monitor and understand your JVM garbage collection very closely. There are several commercial and free tools available to do so. At the minimum, you should enable verbose GC, which will provide all the data that you need for your health assessment
* Keep in mind that GC related problems are unlikely to be caught during development or functional testing. Proper garbage collection tuning will require you to perform load and perform testing with high-volume from simultaneous users. This exercise will allow you to fine-tune your Java Heap memory footprint as per your applications behaviour and load level forecast.

**#4 - Too many or poor integration with external systems**

The next common cause of bad Java EE performance is mainly applicable for highly distributed systems; typical for Telecom IT environments. In such environments, a middleware domain (e.g., Service Bus) will rarely do all the work but rather "delegate" some of the business processes, such as product qualification, customer profile, and order management, to other Java EE middleware platforms or legacy systems such as Mainframe via various payload types and communication protocols.



Such external system calls means that the client Java EE application will trigger creation or reuse of Socket Connections to write and read data to/from external systems across a private network. Some of these calls can be configured as synchronous or asynchronous depending of the implementation and business process nature. It is important to note that the response time can change over time depending on the health of the external systems, so it is very important to shield your Java EE application and middleware via proper use of timeouts.

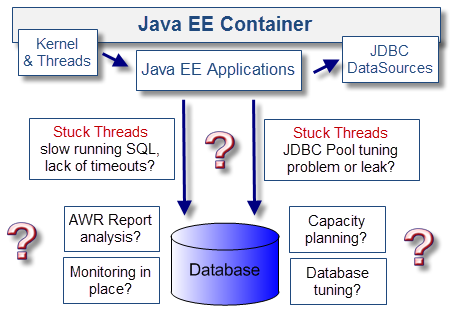
  
Major problems and performance slowdown can be observed in the following scenarios:

* Too many external system calls are performed in a synchronous and sequential manner. Such implementation is also fully exposed to instability and slowdown of its external systems.
* Timeouts between Java EE client applications and external systems are missing or values are too high. This will cause client Threads to get **Stuck**, which can lead to a full domino effect.
* Timeouts are properly implemented but middleware is not fine-tuned to handle the "non-happy" path. Any increase of response time (or outage) of external system will lead to increased Thread utilization and Java Heap utilization (increased # of pending payload data). Middleware environment and JVM must be tuned in a way to predict and handle both "happy" and "non-happy" paths to prevent a full domino effect.

Finally, I also recommend that you spend adequate time performing negative testing. This means that problem conditions should be "artificially" introduced to the external systems in order to test how your application and middleware environment handle failures of those external systems. This exercise should also be performed under a high-volume situation, allowing you to fine-tune the different timeout values between your applications and external systems.

**#5 - Lack of proper database SQL tuning & capacity planning**

The next common performance problem should not be a surprise for anybody: database issues. Most Java EE enterprise systems rely on relational databases for various business processes from portal content management to order provisioning systems. A solid database environment and foundation will ensure that your IT environment will scale properly to support your client growing business.



In my production support experience, database-related performance problems are very common. Since most database transactions are typically executed via JDBC Datasources (*including for relational persistence API's such as Hibernate*), performance problems will initially manifest as Stuck Threads from your Java EE container Thread manager. The following are common database-related problems I have seen over the last 10 years:

*\* Note that Oracle database is used as an example since it is a common product used by my IT clients.\**

* Isolated, long-running SQLs. This problem will manifest as stuck Threads and usually a symptom of lack of SQL tuning, missing indexes, non-optimal execution plan, returned dataset too large, etc.
* Table or row level data lock. This problem can manifest especially when dealing with a two-phase commit transactional model (*ex: infamous Oracle In-Doubt Transactions*). In this scenario, the Java EE container can leave some pending transactions waiting for final commit or rollback, leaving data lock that can trigger performance problems until such locks are removed. This can happen as a result of a trigger event such as a middleware outage or server crash.
* Sudden change of execution plan. I have seen this problem quite often and usually the result of some data patterns changes, which can (for example) cause Oracle to update the query execution plan on the fly and trigger major performance degradation.
* Lack of proper management of the database facilities. For example, Oracle has several areas to look at such as REDO logs, database data files, etc. Problems such as lack of disk space and log file not rotating can trigger major performance problems and an outage situation.

Recommendations

* Proper capacity planning involving load and performance testing is critical here to fine-tune your database environment and detect any problems at the SQL level.
* If you are using Oracle databases, ensure that your DBA team is reviewing the AWR Report on a regular basis, especially in the context of an incident and root cause analysis process. Same analysis approach should also be performed for other database vendors.
* Take advantage of JVM Thread Dump and AWR Report to pinpoint the slow running SQLs and / or use a monitoring tool of your choice to do the same.
* Make sure to spend enough time to fortify the "Operation" side of your database environment (disk space, data files, REDO logs, table spaces, etc.) along with proper monitoring and alerting. Failure to do so can expose your client IT environment to major outage scenarios and many hours of downtime.

**#6 - Application specific performance problems**

To recap, so far we have seen the importance of proper capacity planning, load and performance testing, middleware environment specifications, JVM health, external systems integration, and the relational database environment. But what about the Java EE application itself? After all, your IT environment could have the fastest hardware on the market with hundreds of CPU cores, large amount of RAM, and dozens of 64-bit JVM processes; but performance can still be terrible if the application implementation is deficient. This section will focus on the most severe Java EE application problems I have been exposed to from various Java EE environments.

My primary recommendation is to ensure that code reviews are part of your regular development cycle along with release management process. This will allow you to pinpoint major implementation problems as per below and prior to major testing and implementation phases.

Thread safe code problems

Proper care is required when using Java synchronization and non-final static variables / objects. In a Java EE environment, any static variable or object must be Thread safe to ensure data integrity and predictable results. Wrong usage of static variable for a Java class member variable can lead to unpredictable results under load since these variables/objects are shared between Java EE container Threads (e.g., Thread B can modify static variable value of Thread A causing unexpected and wrong behavior). A class member variable should be defined as non static to remain in the current class instance context so each Thread has its own copy.

Java synchronization is also quite important when dealing with non-Thread safe data structure such as a java.util.HashMap. Failure to do so can trigger HashMap corruption and infinite looping. Be careful when dealing with Java synchronization since excessive usage can also lead to stuck Threads and poor performance.

Lack of communication API timeouts

It is very important to implement and test transaction (Socket read () and write () operations) and connection timeouts (Socket connect () operation) for every communication API. Lack of proper HTTP/HTTPS/TCP IP... timeouts between the Java EE application and external system(s) can lead to severe performance degradation and outage due to stuck Threads. Proper timeout implementation will prevent Threads to wait for too long in the event of major slowdown of your downstream systems.

Below are some examples for some older and current APIs (Apache & Weblogic):

|  |  |  |  |
| --- | --- | --- | --- |
| **Communication API** | **Vendor** | **Protocol** | **Timeout code snippet** |
| commons-httpclient 3.0.1 | Apache | HTTP/HTTPS | HttpConnectionManagerParams.setSoTimeout(txTimeout); // Transaction timeout  HttpConnectionManagerParams.setConnectionTimeout(connTimeout);// Connection timeout |
| axis.jar (v1.4 1855) | Apache | WS via HTTP/HTTPS | *\*\*\* Please note that version 1.x of AXIS is exposed to a known problem with SSL Socket creation which ignores the specified timeout value. Solution is to override the client-config.wsdd and setup the HTTPS transport to <transport name="https" pivot="java:org.apache.axis.transport.http.CommonsHTTPSender"/> \*\*\**  ((org.apache.axis.client.Stub) port).setTimeout(timeoutMilliseconds); // Transaction & connection timeout |
| WLS103 (old JAX-RPC) | Oracle | WS via HTTP/HTTPS | // Transaction & connection timeout  ((Stub)servicePort).\_setProperty("weblogic.webservice.rpc.timeoutsecs", timeoutSecs); |
| WLS103 (JAX-RPC 1.1) | Oracle | WS via HTTP/HTTPS | ((Stub)servicePort).\_setProperty("weblogic.wsee.transport.read.timeout", timeoutMills); // Transaction timeout  ((Stub)servicePort).\_setProperty("weblogic.wsee.transport.connection.timeout", timeoutMills); // Connection timeout |

I/O, JDBC or relational persistence API resources management problems

Proper coding best practices are important when implementing a raw DAO layer or using relational persistence APIs such as Hibernate. The goal is to ensure proper Session / Connection resource closure. Such JDBC related resources must be closed in a finally {} block to properly handle any failure scenario. Failure to do so can lead to JDBC Connection Pool leak and eventually stuck Threads and full outage scenario.

Same rule apply to I/O resources such as InputStream. When no longer used, proper closure is required; otherwise, it can lead so Socket / File Descriptor leak and full JVM hang.

Lack of proper data caching

Performance problems can be the result of repetitive and excessive computing tasks, such as I/O / disk access, content data from a relational database, and customer-related data. Static data with reasonable memory footprint should be cached properly either in the Java Heap memory or via a data cache system.

Static files such as property files should also be cached to prevent excessive disk access. Simple caching strategies can have a very positive impact on your Java EE application performance.

Data caching is also important when dealing with Web Services and XML-related APIs. Such APIs can generate excessive dynamic Class loading and I/O / disk access. Make sure that you follow such API best practices and use proper caching strategies (Singleton, etc.) when applicable. I suggest you read [JAXB Case Study](http://javaeesupportpatterns.blogspot.com/2011/09/jaxbcontext-performance-problem-case.html) on that subject.

Excessive data caching

Ironically, while data caching is crucial for proper performance, it can also be responsible for major performance problems. Why? Well, if you attempt to cache too much data on the Java Heap, then you will be struggling with excessive garbage collections and OutOfMemoryError conditions. The goal is to find a proper balance (*via your capacity planning process*) between data caching, Java Heap size, and available hardware capacity.

Here is one example of a problem case from one of my IT clients:

* Very poor performance was observed from the Weblogic portal application.
* Data caching was implemented to improve performance with initial positive impact.
* The more products they were adding in their product catalogue, bigger data caching requirements and Java Heap memory resulted.
* Eventually, the IT team had to upgrade to 64-bit JVM with 8 GB per JVM process along with more CPU cores.
* Eventually, the situation was not sustainable and design had to be reviewed.
* The final solution ended up using a distributed data cache system, outside the Java EE middleware and JVM via separate hardware.

The important point to remember from this story is that when too much data caching is required to achieve proper performance level, it is time to review the overall solution and design.

Excessive logging

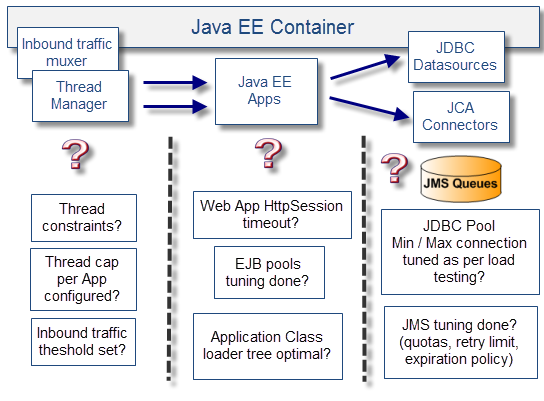
Last but not the least: excessive logging. It is a good practice to ensure proper logging within your Java EE application implementation. However, be careful with the logging level that you enable in your production environment. Excessive logging will trigger high IO on your server and increase CPU utilization. This can especially be a problem for older environments using older hardware or environments dealing with very heavy concurrent volumes. I also recommend that you implement a "reloadable" logging level facility to turn extra logging ON / OFF when required in your day to day production support.

**#7 - Java EE middleware tuning problems**

It is important to realize that your Java EE middleware specifications may be adequate but may lack proper tuning. Most Java EE containers available today provide you with multiple tuning opportunities depending on your applications and business processes needs.

Failure to implement proper tuning and best practices can put your Java EE container in a non-optimal state. I highly recommend that you review and implement proper Java EE middleware vendor recommendations when applicable.

Find below a high-level view and sample check list of what to look for.



**#8 - Insufficient proactive monitoring**

Lack of monitoring is not actually "causing" performance problems, but it can prevent you from understanding the Java EE platform capacity and health situation. Eventually, the environment can reach a break point, which may expose several gaps and problems (JVM memory leak, etc.). From my experience, it is much harder to stabilize an environment after months or years of operation as opposed to having proper monitoring, tools, and processes implemented from day one.

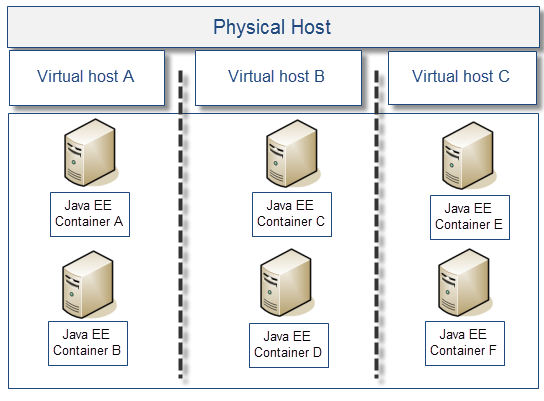
That being said, it is never too late to improve an existing environment. Monitoring can be implemented fairly easily. My recommendations follow.

* Review your current Java EE environment monitoring capabilities and identify improvement opportunities.
* Your monitoring solution should cover the end-to-end environment as much as possible; including proactive alerts.
* The monitoring solution should be aligned with your capacity planning process discussed in our first section.

**#9 - Saturated hardware on common infrastructure**

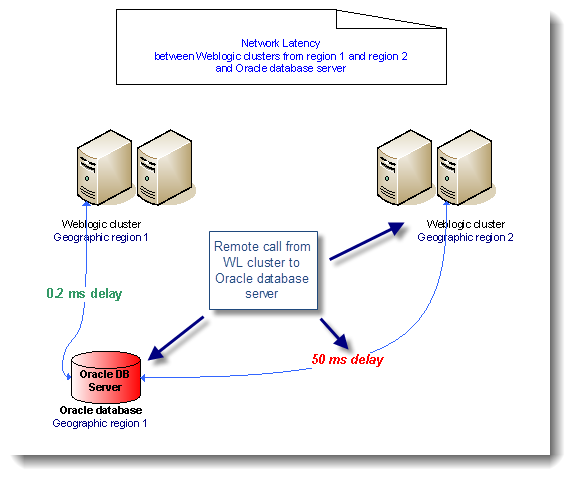
Another common source of performance problems is hardware saturation. This problem is often observed when too many Java EE middleware environments along with its JVM processes are deployed on existing hardware. Too many JVM processes vs. availability of physical CPU cores can be a real problem killing your application performance. Again, your capacity planning process should also take care of hardware capacity as your client business is growing.

My primary recommendation is to look at hardware virtualization. Such an approach is quite common these days and has quite a few benefits such as reduced physical servers, data center size, dedicated physical resources per virtual host, fast implementation, and reduced costs for your client. Dedicated physical resources per virtual host is quite important since the last thing you want is one Java EE container bringing down all others due to excessive CPU utilization.



**#10 - Network latency problems**

Our last source of performance problems is the network. Major network problems can happen from time to time such as router, switch, and DNS server failures. However, the more common problems observed are typically due to regular or intermittent latency when working on a highly distributed IT environment. The diagram below highlights an example of network latency gaps between two geographic regions of a Weblogic cluster communicating with an Oracle database server located in one geographic region only.



Intermittent or regular latency problems can definitely trigger some major performance problems and affect your Java EE application in different ways.

* Applications using database queries with large datasets are fully exposed to network latency due to high number of fetch iterations (back and forward across network).
* Applications dealing with large data payloads (such as large XML data) from external systems are also exposed to network latency that can trigger intermittent high-response time when sending and receiving responses.
* Java EE container replication process (clustering) can be affected and put at risk its fail-over capabilities (e.g., multicast or unicast packet losses).

Tuning strategies such as JDBC row data "prefetch", XML data compression, and data caching can help mitigate network latency. But such latency problems should be reviewed closely when first designing the network topology of a new IT environment.

I hope this article has helped you understand some of the common performance problems and pressure points you can face when developing and supporting Java EE production systems. Since each IT environment is unique, I do not expect that everybody will face the exact same problems. As such, I invite you to post your comments and share your views on the subject.