Advanced Tomcat Configuration and Performance Tuning

Table of Contents

1 Introduction

Tomcat (http://jakarta.apache.org)is a servlet container developed under the Apache license. Its primary goal is to be an accurate implementation and serve as a reference implementation of the Servlet and JSP APIs, and also to be a quality production servlet container. Tomcat works as a standalone server or as a helper for Apache and other major web servers (IIS, NES, AOLServer).

This paper discusses lessons learned working on tomcat, but it applies in other projects where performance is important. It explains how tomcat is designed with respect with performance and how it can be tuned, from both a developer and deployment perspective. This document should be used in conjunction with the other tomcat documentation - the user guide, internals, configuration guides.

Most of the information applies to any server-side java application, and is useful to all Java developers interested in performance.

(This document was originally written for ApacheCon - it uses and will be used as part of tomcat documentation http://jakarta.apache.org/cvsweb/index.cgi/jakarta-tomcat/src/doc.)

2 Tomcat Implementation - Lessons Learned

Tomcat 3.0 had a number of problems, and performance was the least important. The code was stable, but very complex and hard to understand, and all the changes required to implement new specifications made by many people had a significant effect in maintainability .

The first priority was code refactoring. Tomcat 3.1 focused mostly on cleanup and and refactoring, but it also showed a speed improvement (even if that wasn't the main goal). For 3.2 one of the goals was to improve the performance, and we learned a lot from that.

2.1 Cleanup and Refactoring

The simple re-arrangement of the code and better modularization are very important for performance. The magic solution for tomcat was the "Strategy" pattern: encapsulate algorithms and replace them when possible (see tomcat documentation for details).

In tomcat almost all expensive operations are delegated to modules (Interceptors). That helps the code readability and maintenance, but it also allows clean optimizations:

you can isolate particular (simpler) sub-problems

you can test and compare various solutions

no pressure - the original code is still there, as a backup.

80/20 - as long as the ugly code is isolated, you can focus on whatever shows up in profiler

it's easier to profile.

This is also one of the places where you don't trade "readability" vs. "optimization".

For example, most of the changes between tomcat3.0 and tomcat 3.1 consists in moving existing code in interceptors based on their function - parsing request, mapping, authentication, etc. In time (with a strong preference toward code stability) everything got rewritten (few times !). We were free to experiment and improve each piece, and we were free to choose what piece to work on.

2.2 Garbage Collection

Very simple profiling of tomcat showed hundreds of objects allocations per request. We used normal java tools (-prof, -verbose:gc) and OptimizeIt. New Request/Response objects, and all the sub-fields, plus dozens of Strings resulted from parsing. The "Facade" pattern used in tomcat's design is doing the magic here - we can have the internal interfaces completely reusable (no Strings or immutable objects in tomcat.core). "Facade" converts the interface of tomcat.core into the servlet interface - the internal interfaces can now use re-usable objects (like MessageBytes).

The profiling shows a lot of large allocation in the buffers - reusing the buffer in JSP had a huge impact on performances. By exposing the buffer in the internal API we'll be able to eliminate this source of garbage - core.OutputBuffer is able to do that and also reuse the converter objects (IMHO it's a very interesting piece, you should take a look at the code!).

The first thing to do was making sure we understand what and how it is created. Tomcat had support for recycling from beginning, but most of it was unused - it used to create a new Thread per request, and it had no pools.

There are 2 simple mechanisms to allow object reuse in a server environment: an object pool or per-thread data. The object pool (SimplePool in tomcat) allows you to put back the objects after you're done. The thread mechanism uses either a ThreadLocal object (JDK1.2 specific, implementation uses a Hashtable lookup) or extending Thread (MyThread, with a localData fields and casts) or by explicitly passing an Object[] to the handler (what we use). All objects follow a simple pattern - they have a "recycle()" method that will reset the state of the object.

Please note that recycling has some security implications (discussed in a separate document) - in tomcat we are "protected" by the use of the "Facade" pattern.

Recycling works well when the objects have a clean life-cycle - we know we're done with all request-related objects when the connection is closed.

We used both mechanism - the excellent thread pool written by Gal Shachor, and a general pool. The pool requires synchronization, while thread pool allows use of per-thread data (with a slightly more complex interface). In reality we found little difference, with thread data being a bit faster and not very hard to use.

By just reusing the main objects we more than doubled the performance. What's even more important - we reduced the MAX response time. (average response is important, but the top response time can mean lost customers).

In general, it is possible to set the goal at 0 objects per request as tomcat-related overhead. All objects involved in request processing are or should be reusable, and altering the API to make sure we use only reusable objects is well justified. Even if VM gets better and better with GC, the cost of GC can never be 0.

I can't stress enough how important it is to keep memory usage under control in a server environment: yes, GC is faster and faster, but allocation and freeing memory will never be free. What's special for a server is the number of concurrent requests - 80+ threads is not uncommon. Object allocation requires a heap lock in many VMs, and even when incremental GC is used the load on the server may prevent it from working as expected - resulting in long response times and freezes.

2.3 Buffers

One feature that didn't get into 3.2 was the OutputBuffer. The code required many changes in important classes and we felt stability is more important than performance. The OutputBuffer is nothing more than a byte[] with all the methods to add/remove data. It also supports storing chars or other primitive types (note the special case for StringBuffer that avoids toString() conversion - and leave StringBuffer recyclable). We use a special mechanism to reuse the char->byte convertors.

The "chained" streams are great for general-purpose programming, but a lot of data suggest it's not the perfect choice for server performance. No-copy input/output is a common goal in high-performance servers, and it also reduces the memory footprint.

Buffers are a very tricky aspect of the Servlet API, and providing a distinct class and API (de-coupled from Stream/Writer) will help sort this out.

This may also play an important role in JNI mode, and helps improving char-byte conversions.

2.4 Charset conversion (bytes and chars)

This will be another important factor for tomcat 3.3, and is yet another win-win for performance and features.

The 2 important classes are MessageBytes and OutputBuffer.

For input, the charset of the request is unknown until Content-Type header is parsed. The old way of reading the request used a lot of Strings (see the discussion about String) and the problem is the Strings were created with the wrong

encoding (you need an encoding to create a String from the bytes - no encoding means platform's default). MessageBytes (described later) will (hopefully) solve this problem and also avoid a lot of object allocation - not all headers and properties are actually used by all servlets. It also allows optimized convertors.

The conversion between bytes and chars is very simple in Java - and that's great in most cases. If you have a byte[] and encoding - you just call "new String(bytes, enc)". The problem is that you have no way to control what happens inside - and what happens is (at least in some VMs) a possible 8K buffer allocation. The buffer is very important for convertor's performance - but it's just GC that hurts a lot in a loaded server environment.

Internally, OutputBuffer uses some tricky code that reuses the convertor. In time we'll replace the hack with specialized convertors - that's what Xerces is doing.

For MessageBytes we provide special case for ASCII (headers are required to be ASCII, and most - but not all values in a request are also ASCII), and we delay the conversion until it's actually needed (this also means only the code that uses the string values is paying the price)

2.5 Module communication - Notes

A very interesting piece is the mechanism used to keep module-specific state in request, context and context manager. Most servers provide such a mechanism using key/value pairs, and either hashtable or linear searching to store/retrieve the values.

In tomcat we used a very common optimization technique - moving constant code outside the loop. Instead of using a hashtable to store the values with a String key we pre-compute the hash code and index when the component is initialized. All set/get methods take an int key and have O(1) cost. We separate the constant code - computing the hash of the key and the index, while still using descriptive and extensible key/value pairs.

The "Notes" are simple Object[], where information is stored and retrieved in O(1) (array access time). Instead of hard-coding the indexes (0=MY_OBJECT, 1=YOUR_OBJECT) we use normal keys as in a Hashtable. The difference is that instead of calling get(String key), we call getNoteId(key) and then get(int id). The first method (getNoteId) is called in init() - or outside of the critical path. This is exactly what happens in a hashtable (or even better - we don't have to deal with hashing conflicts), except that the API exposes the 2 steps typically used in hashtable implementation.

Notes are not a major factor in performance - the number of accesses is small compared with the rest of the request overhead, but a good idea that may be very useful in some cases.

2.6 Specialized data structures - MimeHeaders

Another place where the general-purpose java.utils can be replaced with specialized components. Access to headers (and parameters - we plan to use a similar/same object for parameter storage) is a very common operation and have very special characteristics. By using a class that is tuned to the use-cases we can improve a lot the performance compared with a simple Hashtable.

For example we know most headers will have a single value, but some may have multiple values, we know the average size, we know the header name is ASCII and we know a number of headers will repeat often. We also know a number of headers are more likely to be read (or will always be read- ContentLength, Cookies) - so those operations can be optimized. We also know some fields will be converted to int/date and we can cache the conversion result.

2.7 String overuse - MessageBytes

Almost all texts on java performance mentions the high cost of using String. The fact that it's immutable is vital for security, and all public APIs are using it.

Another pattern used in tomcat's design helps a lot resolving this - the Facade allows tomcat to use MessageBytes internally while providing String based interface for the servlets. The Facade is also very important for security, by isolating the internal implementation code.

One interesting (and I hope well known) fact is that StringBuffer can't be reused if toString() method is called (the buffer will be owned by the String, and setLength(0) will create a new buffer). We do use and re-use the StringBuffer, but we make sure toString() is never called (it's not hard to spot this - it shows very clearly on any memory profiler).

2.8 80/20 and the critical path

Startup time and all async operations are not very important in the server operation. We have to look at the "critical path", what happens between the moment a request is received and the moment when the servlet service() is called, plus all the calls to the servlet API that tomcat implements.

Maintenance operations do matter as memory overhead - especially those that happen per request, like logging or session keep-alive. They also matter as the load increases - if the thread has lower priority the maintenance may not happen, and if not it will directly affect the response time - even if it's considered a background thread.

2.9 Optimization vs. code clarity

We all know the famous "Premature optimization is the root of all evil ". How does it applies to tomcat?

The original design was based on a number of simple design patterns and the experience with previous web servers. Tomcat 3.0 is based on JSWDK (the reference implementation of the servlet api) and 3.1 was a major refactoring. The real optimization started in 3.2 - it's hard to say it's "premature".

Choosing the right patterns at the beginning is probably the most important factor in tomcat performance, but again it can hardly be considered "optimization".

We also found that most of the performance improvements come from simplifying the code and modularization - with only minor gains from low-level code hacking (final methods, class vs. interface cost, longer methods to avoid method call overhead - there is a whole programming style with well-known examples).

2.10 Other factors

Request parsing and matching. So far the mapping doesn't qualify as a hotspot, but that's only because other components need tuning. As the "bad" code is cleaned up this will probably show in profilers. As Invoker (the default "/servlet/*" mapping) is no longer available (not part of the spec) we can expect a big increase in the number of mappings per web application.

Web server adapter. So far JNI (tomcat runs in-process, the communication with the server provided by simple JNI method calls) proved to be the fastest solution, with AJP13 (a tcp-based communication protocol optimized for tomcat's needs) having promising performance. Standalone http is used in many configurations, and we know that it can be extremely fast (since other java-only servers did that before). We have to focus on all 3 components. Ajp12 is the most stable, but it's better to leave it as it is (i.e. not try to change it).

Servlet API optimization. Servlet API defines a number of method that a servlet can use, and we need to make sure the implementation is as efficient as possible. It is important to identify the most frequent calls (like getParameters(), session, etc.) and spend the time on them.

Background activity. Tomcat needs a number of threads that will monitor and maintain it. For example it needs to expire sessions, detect changes, maintain pools, etc. While most of it happens in background, under high load it became a factor. Even more importantly is the garbage that it generates, because it will affect tomcat

even with average load (it is possible to use lower priority for maintenance threads, but GC affects all tomcat activities).

Caching. Of course, but keep in mind that this conflicts sometimes with scalability (the memory usage increases - and remember the servlets and jsps may use a lot of code memory too), and there are better ways to do that (reverse proxy / accelerators, etc.). It may be better to let the main web server do the caching - it have far better I/O system (by using OS-specific features).

Memory footprint. As with any server, if it doesn't have enough memory the performance will quickly degrade. The things are worse because of GC, and caching is also playing a role. It's important to remember that we're running servlets - and the servlet developer is less likely (and has less control) to do optimizations.

Algorithms. This is very important - and it's related with the refactoring and the use of Strategy. For example parsing the request and matching it against mapping rules is probably where it'll make a big difference to use an advanced Tree-based search algorithm or even a specialized one.

JNI overhead. There is a big difference between a JNI and a normal method call. While this will probably change with the VM, it is important to minimize the number of native invocations and use a big-enough granularity. The cost of a TCP round trip is even bigger, so both JNI and AJP13 will do that.

Code duplication. In tomcat 3.2, all the request processing is duplicated in the web server and in tomcat. This is a clear waste, and have to be eliminated.

Large servers (100+ web applications). The mapping tables will be very big and will use a lot of memory, while most applications are not accessed most of the time. We need to keep the memory usage under control - for example by dynamic loading and unloading of unused apps. (since Invoker is no longer part of the spec, most webapps will have to define all mappings explicitly - resulting in the very large mapping tables)

Session thread usage. The current session implementation use 1 thread per web application - this is a problem if many web apps are installed. A web server/servlet container is already a very thread intensive application, and the current use is just plain waste and can push the OS to the limits. Note that it's not possible to use only one thread for all sessions - a "bad" application may hang in the notification method. (one maintenance thread + a thread pool for callbacks).

Optimized char->byte convertors for other charsets. Xerces is a good source - it seems the code just needs to be cleaned up and adapted. AUC (a global collection of Apache Utility Classes where we could share code between projects) would be great for that.

Date The current time formatting is very slow and generates a lot of garbage. It is very important to minimize the use in the request (for example by using a special AJP13 extension and send the long date to the web server, where a specialized formatter can be used)

2.11 Non-issues

Method call overhead (too many hooks in the chain). Most JITs are able to eliminate "dead" code, and a method call is around 0.2 us. (in 3.3 all empty calls will be eliminated - again as part of a refactoring of interceptors, intended to improve readability)

State maintenance and **inter-module communication**. The current note mechanism reduce the request state to a indexed array access - faster than mechanisms used by Apache or NES for the same function (key/values, with hash or linear searching). We can also add explicit get/set if a field is of common interest for modules.

Low level java VM optimizations (class vs. interfaces calls, etc.). I have no clear data (as I never tried to do that), but I expect it to actually reduce the performance - as the code will be less readable and more important optimizations will become impossible.

3 Deployment and configuration

3.1 Server Setup

One of the goals is to make tomcat a re-usable component that can be embedded in applications and web servers. There are 3 major use-cases:

Tomcat standalone (as a web server). This will work for small web sites, but it's mostly intended for development and testing. The current performance is not bad - and if your site hosts only servlets and it's not big you may be ok with it, but for a production server you may need a real server. The HTTP implementation is minimal (and 1.0 - we are working to port-back the 1.1 implementation from a different container) and is not likely to ever match a "real" and sophisticated server like Apache (and it's not our goal to do that - we are implementing a servlet container, not a web server!).

Tomcat embedded in applications to provide Web Server and Servlet support. In most cases tomcat will work as a standalone server (using the internal HTTP implementation), but it is possible to link the application with a real web server later. This allows applications to provide web-based services (like configuration, access to data) and also allow use of servlets in applications (with or without the use of HTTP).

Tomcat + "real" web server. Tomcat can be integrated with Apache 1.3, Apache 2.0, NES, IIS and AOLServer , and based on the current experience it is very easy to write connectors to other servers. This is the preferred solution for a production site. It allows you to add servlet capabilities to existing sites with minimal changes in the site architecture. Most web servers are highly tuned for each platform, using OS-level calls and optimizations.

Current web servers (IIS, Apache, NES, AOL) provide a very good infrastructure of modules and tools. Even if we can write very fast code in Java, using the time-proven server modules will insure speed and stability. It is also very hard to match all the optimizations that are done in the native web servers, some of them having heavy dependencies on the OS and platform they run on.

3.2 Server Adapters

In order to work with a web server tomcat uses a "server adapter", that provides the communication mechanisms. Tomcat doesn't require any particular protocol - the implementation is just a normal tomcat interceptor.

Most of the development was done in the jk (src/native/mod_jk) adapter, that implements a 2-layer architecture. The first layer is the server module, with implementations for Apache 1.3, 2.0, NES, IIS, AOL. The second layer is a protocol used to transmit the internal server representation of the request to tomcat and get back the response. All this uses a collection of portable code (that will eventually be replaced with APR - the Apache Portable Runtime - the current code tries to make this change as easy as possible by using similar code).

A number of different protocols are available, and to make things more complicated we also support the original mod_jserv adapter and the standalone http interceptor.

This follows the same "Strategy"pattern - we don't know yet what is the best protocol, and it is very possible that different protocols will act better in different circumstances.

JNI (implemented as part of mod_jk). This is my favorite, and provides the best speed - tomcat runs in the same process with the web server. It also provides great opportunities for integration of Interceptors and native modules (filters, SAFs). It doesn't work with non-threaded servers and can't be used with multiple computers. It is best for webapps that have low computing requirements and lot of input/output, or for servers using "big" hardware (few powerful computers as opposed to farms of small computers). It is also perfect for applications that don't maintain state (or use their own mechanisms). It will be a perfect solution when hardware/software load-balancing will support servlets.

AJP12. This is the most stable protocol. It's implemented in mod_jserv and mod_jk, and it is around and tested for many years (in tomcat and Jserv). It's also very simple and maintainable.

AJP13. Probably this will be the future protocol-of-choice. It's an improved 12, with connection re-use and callback support. The code is stable and can be used, and most of the development will focus on this protocol.

HTTP. This is a particular case, it is mostly used for standalone tomcat (few people used it with mod_rewrite and mod_proxy in apache). The protocol is tuned and have decent performance. HTTP1.1 implementation is not ready (it has to be back-ported).

3.3 Setup Examples

Apache 1.3 + Ajp12 + Tomcat.
: This is the most stable configuration integrating Apache as a web server and tomcat as a servlet container

Apache 2.0 / IIS / NES / AOL + Ajp13 + Tomcat
: Apache 2.0 and Ajp13 are considerably faster and this is probably the future configuration of choice, especially with load-balanced servers. Running tomcat as an external process allows farming and is probably safest.

Apache 2.0 /IIS / NES / AOL + JNI + Tomcat
: For sites with applications that need the smallest response time and if the computing requirements do not requires farming (or farming is done at a higher level) JNI is the fastest way of communication between tomcat and Apache. It is possible to mix JNI and Ajp13 with WebApp granularity

Standalone tomcat
: This configuration is used when tomcat is embedded in applications to provide web services. This is not a "high performance" setup, since we can't re-invent all the optimizations performed by Apache or other web servers.

3.4 Tomcat configuration

Tomcat is composed from a core and individual components (interceptors). In order to configure tomcat you need to specify the interceptors you want and to set up each individual component. Each component have reasonable defaults, but it's normally tuned towards a development setup, with most features enabled.

The version of tomcat released under jakarta.apache.org is set up using "server.xml" - tomcat may be embedded in applications and use the application native configuration mechanisms. In general, all components are independent of the configuration subsystem used.

3.5 Tuning

The default Tomcat configuration enables most features and is intended for servlet developers. We assume there are more people using tomcat to develop and test web applications, and most "production" sites will need to review and customize the configuration anyway. We also assume web site administrators have more experience configuring and tuning software.

In my experience, using a better VM provide the most benefit. The JDK implementations may reach a limit (they already run at a speed comparable with C), but for now hotspot or a good JIT will make a huge difference over older VMs.

It is also very important to choose the best combination of web server, protocol (ajp, jni), application distribution on available servers (in case you have a farm of servers). The application is probably the decisive factor, and choosing a deployment configuration based on application will give you most benefits. You may choose multiple "profiles" - for example use both JNI and AJP13, and deploy all I/O intensive, low CPU applications in-process and all compute-intensive applications on a pool of tcp-connected servers.

You have 2 choices for high-traffic applications - either use a generic pool and deploy all applications on the pool, or use dedicated servers with fewer webapps. For critical applications it's better to use dedicated servers.

A number of configuration factors will affect your performance:

VM. This is by far the most important external factor. Use the best VM for your machine, use the machine that has the best VM! Right now JDK1.3 (where available) seems to be significantly faster - with hotspot or IBM jit. You need to specify the JAVA_HOME environment variable before starting tomcat. We strongly recommend 1.2 or higher - many features will work only with Java2 (security for example).

Reloading. Just don't forget to turn it off in production sites. In order to disable reloading you need to specify ' reloading="false" ' in the context attributes. In tomcat 3.3 reloading is redesigned and have a lower overhead (it no longer happens for every request, but only at regular intervals). It is still using cycles and disk accesses, and may affect a loaded server.

Logging. In a production site you'll probably use a web server native logger. The internal logger should only be used for exception reporting, not for access logs. In fact the logging is an important factor for native servers too - and it's likely to be highly optimized. For standalone tomcat you can use a custom logger. You can set 'verbosity="FATAL"' and 'timestamps="no"' or 'timestamps="msec"'.

Security. Turning policy-based security on does reduce the performance - but most of the time it's better to trade performance for security. I strongly recommend leaving it on and making sure your applications are able to run in a sand-boxed environment. You disable the security by removing the policy interceptor from the configuration.

OS tuning. Same as for normal web servers - make sure you have enough file descriptors (Ajp12 will use an additional connection per request), make sure you have enough memory. The rules used to prepare a server for Apache should work for tomcat too.

Profile your application (OptimizeIt ?). Tomcat shouldn't take more than 20..30% of the total time - so most tuning should and optimizations should be in your code. This is different from a normal web server, where most of the content is static. Tomcat is used mostly for dynamic content (servlets, java), with the static files served by the real web server.

Test under load. Running your application from a browser is not enough - you need to make sure the server load match your expected number of users. Use ab ("/usr/local/apache/bin/ab") to test individual pages. We are working on improving the <gtest> ant extension (that is used in watchdog and for tomcat testing) to allow more advanced tests (sessions, etc). Any tool that simulates the load is good (including perl)

Deployment configuration. You may need to use a pool of servers. You can use load balancing or use dedicated servers for each application (that's also good for security and stability). The number of threads (concurrent requests) per tomcat instance should depend only on your application - if the overhead of tomcat is more than x % you should try a different servlet container.

Java compiler (jikes, javac, javac -O). It's a small increase, but it's free (the JIT is very important and will likely do most of the optimizations). Make sure tomcat is compiled with optimization turned on and debugging turned off. Same for applications - but only on the production server.

Heap size. Make sure have gets enough heap space. (-ms256MB, -mx256MB). There is an interesting tradeoff between performance (heap size < real memory) and preventing "out of memory errors" (when you allow swapping). In most VMs swapping is a very bad idea - memory access is random and continuous (GC), and you'll not get infrequent accessed data to remain swapped-off.