

# **Servlet Performance and Apache JServ**

---

## **ApacheCon 1998**

By Stefano Mazzocchi and Pierpaolo Fumagalli

### **Index**

1	Performance Definition .....	2
1.1	Absolute performance .....	2
1.2	Perceived performance .....	2
2	Dynamic Content Generation .....	2
2.1	Overview of Apache module structure .....	2
2.2	mod_cgi.....	2
2.3	mod_perl.....	3
2.4	mod_php.....	3
2.5	mod_jserv .....	3
3	Servlet Technology.....	3
3.1	Pre-request stage .....	3
3.2	Request stage.....	3
3.3	Final Stage .....	4
4	Apache JServ mode of operation.....	4
4.1	Class loading and caching.....	4
4.2	Multi-threading .....	4
4.3	Request processing .....	5
5	Apache JServ Performance improvements.....	6
5.1	Java Virtual Machine.....	6
5.2	Local Mode .....	6
5.3	Authentication .....	6
5.4	Class auto-reloading .....	6
5.5	Tracing .....	6
6	Future Performance Enhancements .....	7
6.1	Recycle Technology.....	7
6.1.1	Socket connections.....	7
6.1.2	Request handling threads.....	7

## Performance Definition

To analyze performance, we need to define its meaning in this specific client/server context. We identify two different performance approaches, a more general one common to many other execution paradigms, and a more specific one especially defined for client/server models.

### **Absolute performance**

Absolute performance measures the time taken by an application to execute. This definition is well suited for programs with no asynchronous delays (user interaction, network congestion, machine load) and it's used as an index of speed for such contexts. It's easy to understand how this idea of performance may not be used to test a server speed.

### **Perceived performance**

Perceived performance measures the time taken by a server to execute and terminate the client requested action. This definition is well suited for client/server paradigm where the server performance perceived from the client matches this measure. From now on, server performance and speed will be used to define this type of measurement.

## Dynamic Content Generation

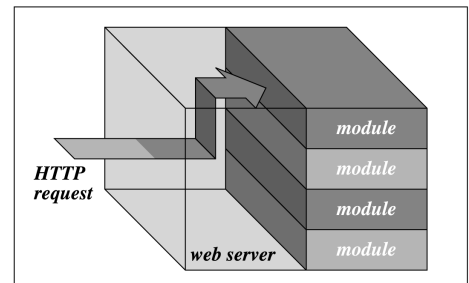
Apache web server was designed to be a modular web server. Modular means that its capabilities could grow adding some so-called "modules". The functionality of Apache is not, like in many other cases, relative only to the core program, but to core and modules.

### **Overview of Apache module structure**

To support modules, Apache has a set of APIs used by module developers to connect to the core program. The core program, then, calls the proper registered modules whenever this is requested by a specific event. Such events could be, for example:

- initialization: a specific procedure in the module program is called when the server initializes, to allocate those resources it may need during its further processing
- authorization: a module could grant, or deny, access to a client for a requested URI
- request: a module could process the request instead of Apache itself
- and others, but less important.

The most important of these is, of course, the request event. When a client connects to Apache web server requesting a URI, the server searches through its modules if one could process this URI giving a response to a client. If this is found then the request is completely handled by the module, which acts, for the client, as the server itself.



Modular web server request handling

### **mod\_cgi**

This is a module handling CGI scripts. A CGI script is an executable program executed by the web server. It gets request parameters and data from its environment variables and from its standard input, then it computes its operations and returns the response through its standard output.

mod\_cgi is called whenever a client request an URI that the web server recognize, from configuration files, to be a CGI script.

The module then executes the program, passing the right environment variables, and returns the program's standard output to the web server's client.

This is a simple, but powerful, way of generating dynamic contents.

### ***mod\_perl***

This is one of the most powerful modules available for Apache. `mod_perl` in fact comes with a full set of APIs for the Perl language, to enable Perl script to act as Apache modules. In fact one could write a Perl script to check URI accesses, or to generate responses to specific client requests.

The power of Perl resides in its wide range of possible applications, in fact Perl APIs were already written to connect to databases, to access system resources, and many others.

Here, like in Java, an interpreter of the Perl language is required; `mod_perl`, then integrates the Perl runtime library into the server module.

### ***mod\_php***

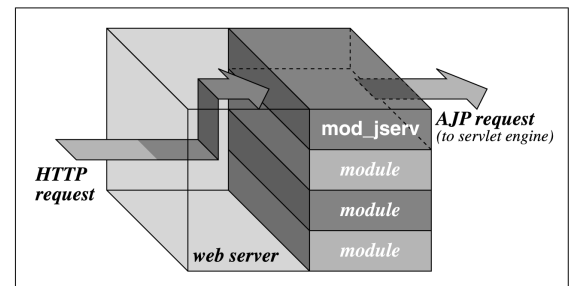
The idea of Rasmus Lerdorf to have a server-side HTML-embedded scripting language became today what usually users call PHP. `mod_php` is called whenever a specific PHP file is requested by a client; the module, at request time, parses and interprets the requested file, sending back the output to the client.

The power of PHP resides in its grammar simplicity (PHP is similar to HTML), and in its wide range of APIs. In fact PHP right now is able to connect to a great number of databases (some of them are Oracle, mSQL, MySQL and ODBC) and has some important features like GIF images creations and more.

### ***mod\_jserv***

Completely different from the other modules, `mod_jserv` does not include a parser (like in `mod_php`) or an interpreter (like in `mod_perl`). Its only task is to convert an HTTP request into an AJP request, send it over the network, wait for an AJP server to execute it, and to return the AJP responses (translated back to HTTP) to the client.

`mod_jserv`, so, does not rely on itself for proper servlet execution, but on the power of the servlet engine, in this case, Apache JServ.



*mod\_jserv request handling*

## **Servlet Technology**

In the servlet specification design, it's well defined the separation between actions performed before and after a client request. In fact, the servlet execution paradigm allows a dynamic content generator to increase its performance because most of the time consuming actions may be performed before the client requests the action.

### ***Pre-request stage***

The servlet engine performs the most heavy and time-consuming operations needed by the servlets. Resource connections (*URL*, *JDBC*, files) and memory allocation are the most common operation performed in this stage.

### ***Request stage***

The servlet engine accepting the request executes the servlet and returns the generated output to the requesting client. The time taken by the requested servlet to terminate its service is the sum of the time taken by the servlet engine to load and run the servlet and the time taken by the servlet to execute. It's the servlet engine's responsibility to provide the servlet with fast APIs in order to reduce total execution time.

### **Final Stage**

In this stage the servlet has the ability to clean up before being destroyed. The servlet should use this stage to close currently open connection, de-allocate memory and free used systems resources.

## **Apache JServ mode of operation**

### **Class loading and caching**

When the servlet engine is started, some servlets (specified in its configuration parameters) are loaded and cached. All other servlets are loaded and initialized when their first request is made. The forced pre-loading feature is used to avoid the big latency that would appear on the first request to non-initialized servlets.

Every time a servlet is initialized, either at startup or after the first request, a cache system is responsible to keep in memory the servlet class and all those classes (even system classes) used during its execution. These caching capabilities allow fast transition from the idle to the executing state, together with faster overall execution.

One of advanced Apache JServ features is servlet autoreloading: this means that when a cached servlet is updated, either changed or removed, it is dumped from memory and reloaded from disk. It must be noted that when a servlet is changed, the state of the whole cache could be unpredictable since other non-servlet classes that are currently cached may have changed. To avoid the risk of possible conflicts due to this state inconsistency of the cache, when a single class is changed, the whole cache is forced to cleanup itself.

To avoid excessive servlet engine overload due to a single class update, each servlet zone has its own class loader. This feature allows, for example, the use of a single servlet engine instance for both servlet development and production serving. In fact, separating the two logical groups of servlets (the final ones and the ones under development) in two different zones allows class reloading to be performed in the zone where it is most needed without causing performance problems on the other zone.

When the zone's class loader is forced to cleanup, every servlet is called to do its cleanup and its destruction stage. For this reason, class reloading should be avoided in production servlet engines where performance and service availability are major issues.

### **Multi-threading**

Apache JServ is a full multi-threaded server and allows multiple requests to be handled concurrently. This means that perceived performance is increased since the latency between the client action and first generated output depends very lightly from the number of concurrently requesting clients. Also, if the underlying JVM, operating system and hardware support multiprocessors, a different processor may execute each thread, thus reducing the load of the server by a factor equal to the number of processors available.

Multi-threading is a very powerful feature, but careful design should be used to avoid state inconsistencies, deadlocks and unpredictable behavior. In fact, Apache JServ uses its own locking mechanism to secure the execution of the servlets to avoid destroying a running servlet during the class cache clean up.

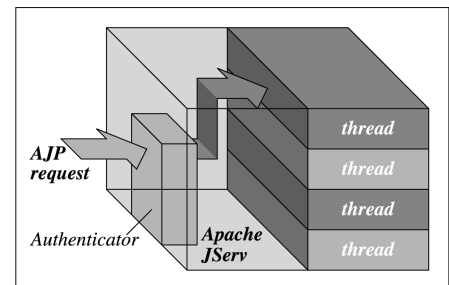
The servlet engine stores only one instance for each servlet and each request handling thread executes the same method concurrently on the same servlet, as specified by the servlet APIs. Since this may be a dangerous thing, the servlet specification provides the servlet developer a way to force the servlet to have a several instances, each one handled by a different thread without concurrency. To do this, a servlet must implement the `SingleThreadModel` interface and it will be guaranteed by the servlet engine that no other thread will run the `service()` method of that servlet until it's done.

### Request processing

These are the operations that are executed by the servlet engine when a request comes to the port it listens to. In this stage, speed it's essential since every operation between the client request and the servlet's start decreases the overall performance of the server.

Here is what happens when the web server recognizes a servlet request:

- 1 The web server dispatches the request to the `mod_jserv` module.
- 2 `mod_jserv` maps the requested `URI` with the correct servlet engine using its configurations, then it connects to it.
- 3 The servlet engine, listening on a port, acknowledges a connection then
  - 3.1 Controls if the `IP` address of the requesting entity is allowed to connect.
  - 3.2 If the point above is passed, the authentication handshake is performed (as specified in the Apache JServ Protocol v1.1)
  - 3.3 If the two above test are passed, the connection is authenticated and gets handled.
- 4 `mod_jserv` assumes the connection is authenticated (to avoid delays) and sends the request to the servlet engine.
- 5 The authenticated connection is passed to a thread for handling.
  - 5.1 Request parameters (environments, headers, and cookies) are parsed and stored into hash tables for later usage.
  - 5.2 Session information is retrieved.
  - 5.3 The servlet's `service()` method is called to pass request and response data and to start the servlet.
  - 5.4 As soon as the servlet generate output, it is sent back to the requesting client.
  - 5.5 When the `service()` method returns, servlet execution is over and the handling thread is destroyed.
- 6 `mod_jserv` listens on the connection, creates an `HTTP` response with data coming from the servlet engine, and sends it back to the client.



Apache JServ mode of operation

## Apache JServ Performance improvements

Now that we have seen in great detail how the servlet engine handles the request, we look closer to what that can be done to increase its performance, both absolute and perceived.

### Java Virtual Machine

Like any other Java application, most of Apache JServ performance is due the underlying virtual machine. It is important to note, however, that since servlet engines are very specific applications, general performance improvements like just-in-time compilation may not increase the speed of the server as whole.

Here is a list of JVM features that influence Apache JServ:

- *Native thread support* always increases the speed of heavily multi-threaded applications like a servlet engine. In fact, green threads are a virtualization used on operating systems that do not have native thread support. Their use adds complexity to the execution and increases the thread creation time.
- *Just-in-time* compilation does always play an important role in Java performance, but may not always increase it. A JIT enabled JVM execute Java code faster than normal bytecode interpretation only when close loops are present and the time taken to just-in-time compile the Java bytecodes into native opcodes is diluted in the, now faster, loop execution. If tight loop presence is negligible, the JIT wastes time compiling a code that will be never executed more than once. Since this behavior may change a lot depending on code structure, it is suggested to perform some execution tests on the servlets to find out if JIT is helpful.

### Local Mode

Since the web server uses a network connection to communicate to the servlet engine, a big performance bottleneck could be the network itself. To avoid performance degradation due to network load or failure, it is suggested to make the two server share the same system and use network loopback interface (IP address 127.0.0.1) for connection.

### Authentication

To avoid intrusion and untrusted servlet execution, each connection must perform an authentication handshake to be able to request the servlet. This procedure increases delay by a network round-trip time (see the `AJP` protocol specification for more information). To eliminate this delay, authentication may be disabled, still maintaining a comfortable security level with the `IP` filter list. It is suggested, though, to disable authentication only when absolutely needed or when other external network protections are in use.

### Class auto-reloading

Since servlet reloading is a feature commonly used only during development, it is suggested to turn it off on those servlet zones where servlet content is constant.

### Tracing

Apache JServ provides extended logging and tracing capabilities, but they are mostly used during internal debug by its core developers or by power users willing to know its internal workings in great detail. Since logging and string creation are very expensive tasks for a Java program, especially when a high tracing detail is used, it is suggested to leave tracing off (default settings) on production servers, or at least, to enable only those tracing channels (*exceptionTracing*) that generate only small and asynchronous data.

## Future Performance Enhancements

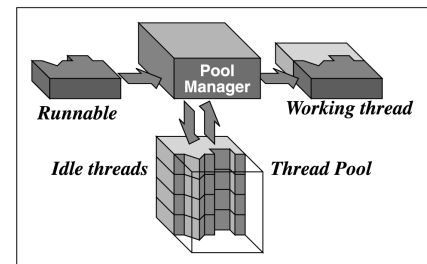
Since Apache JServ reached a beta level of stability not long ago (the first final release is not yet available), stability and not performance was the main issue during its development, even if the servlet engine turned out having good performance.

Future releases will address the need for both faster request execution and lower memory consumption working on what have been identified as major performance bottlenecks.

### Recycle Technology

Garbage collection represents one of Java main features and it's a great help for software developers. Unfortunately, garbage collection is an expensive process (up to 20% of total JVM CPU cycles), especially in multi-threaded servers where idle time may be low and garbage generation high.

To eliminate this bottleneck, we defined a new paradigm called Recycle Technology that abstracts the need of recyclable objects to lower the burden of garbage collection. Exactly like in real life, the garbage collector cleans up those objects that are created during normal execution and go bad, while recyclable objects will be cleaned, stored and reused. This "ecological" treatment reduces the power (CPU cycles) needed to create and destroy a recyclable object, thus increasing the application's performance.



Thread Recycle Technology

Here is where the major recycling effort will be focused on:

### Socket connections

Current communication protocol (**AJP** version 1.1) implies a new socket connection to be created for each request and an authentication procedure to be performed. The new **AJP** version 2.1 protocol (defined but not yet implemented) will guarantee a keep-alive behavior and allow socket recycling, thus removing both the creation and authentication delays from the request time.

### Request handling threads

Another significant impact on perceived performance is thread creation. Again, the use of recycle technology can eliminate the need for on-fly creation using a pool of idle threads that are created at server startup.