

Advanced Apache JServ Techniques

ApacheCon 1998

By Stefano Mazzocchi and Pierpaolo Fumagalli

Index

1	Abstract	2
2	Glossary	2
	2.1 Servlet	2
	2.2 Servlet Engine	2
	2.3 Servlet Repository.....	2
	2.4 Servlet Zone	2
3	The Servlet life cycle	2
	3.1 Initialization.....	2
	3.2 Execution.....	3
	3.3 Destruction	3
4	Apache JServ “three-tier” model	3
	4.1 mod_jserv	4
	4.2 Apache JServ	4
5	Apache JServ scalability	5
	5.1 Low resource usage of the mod_jserv module.....	5
	5.2 Network connection between the two sides	5
	5.3 Local mode	5
	5.4 Remote mode	5
	5.5 One to One connection	5
	5.6 One to Many connection	6
	5.7 Many to One connection	6
	5.8 Many to Many connection	6
6	Security issues	6
	6.1 External security	6
	6.1.1 Intrusion.....	6
	6.1.2 Denial of service	7
	6.1.3 Packet sniffing.....	7
	6.1.4 Packet spoofing	7
	6.2 Internal security	7
	6.2.1 Servlet sandboxing.....	8
	6.2.2 Servlet zones	8
	6.2.3 Current security model	8
7	Configuration Examples.....	8
	7.1 Local Mode configuration example	9
	7.2 Remote Mode configuration example	9

Abstract

Analyzing the problem of adding servlet support to a web server, several solutions have been developed and many software packages are already available to achieve this result. Apache JServ is one of these solutions and differs from others because its main target is the use on the widest range of network environments. Here, we'll introduce Apache JServ advanced features and its inner workings to allow both network administrators and servlet writers to achieve best results, both in network design and servlet development.

No particular knowledge will be required in order to understand the covered issues, even if a prior knowledge of both the Java language and network design may be helpful.

Glossary

Before going on, we need to introduce some keywords and concepts that will be used throughout the discussion.

Servlet

A servlet is a Java server side application that runs inside a network service, such as a web server. It responds to requests from clients, accepting client input and dynamically generating output. For example, a database querying servlet may receive a client's query, run it against the connected database, process obtained data, and return formatted output to the client.

Servlet Engine

A servlet engine is a server application, or part of it, that executes a servlet, forwarding client's request data to the requested servlet, and servlet response back to client.

Servlet Repository

A servlet repository is a collection of servlets and may either be a directory or an archive, such as a zip or jar file.

Servlet Zone

A servlet zone defines the sandbox where servlets live during their life cycle. A zone defines the security environment where the executing servlet is restricted and allows context separation between different logical groups of servlets. A servlet zone is the servlet engine equivalent of a web server's virtual host.

The Servlet life cycle

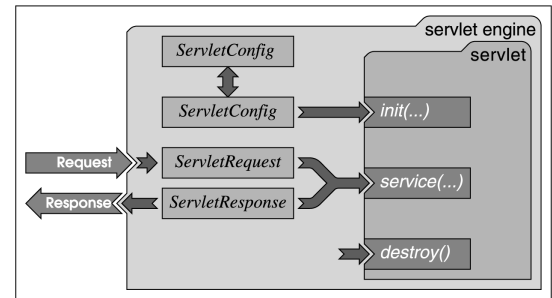
Before analyzing in deep how Apache JServ works, we introduce how a servlet engine runs a generic servlet. The *JavaSoft Servlet API* has specified this execution paradigm and a servlet's life cycle was separated in three different stages:

Initialization

In this first stage the servlet engine loads the servlet's ".class" file in the Java Virtual Machine memory space, instances (converts the loaded file into a valid object) and initializes it. The initialization stage may be used by the newly loaded servlet to allocate those resources it will need later on during execution.

Execution

Whenever a servlet request is made to the servlet engine, the requested servlet instance is already present in the servlet engine memory space, or, if not present, the initialization stage is performed. With the servlet request, the servlet engine receives all request parameters that are used to construct a **ServletRequest** object. This object will be used by the servlet to gather all needed information about the request made by the client. Another object, **ServletResponse**, contains all the hooks the servlet uses to return its output to the requesting client. This execution stage will take place as many times as many requests are handled by the servlet engine.



The servlet execution stages

Destruction

In this latest stage a servlet is requested to cleanup any resource allocated during initialization and to do a graceful shutdown.

These three stages are controlled by the servlet engine that manages them by calling the appropriate methods in the servlet class (the constructor method and **init()** during initialization, **service(ServletRequest, ServletResponse)** in execution and **destroy()** in destruction). These methods represent the execution hooks comparable to the **main()** method of an application or to the **start()** and **stop()** methods of an applet.

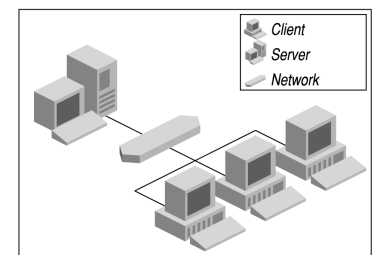
Apache JServ “three-tier” model

Running a servlet is a similar task to the execution of a *CGI* or *PERL* script. In fact like these two need an interpreter (a shell in case of a generic *CGI* shell script or the *PERL* language interpreter), also the servlet, compiled into Java bytecode, needs a Java Virtual Machine to be interpreted.

The significant difference between a shell or *PERL* language interpreter and a Java Virtual Machine is spawning time. The first ones can be spawned in a limited time and use a very little memory (*GNU bash 2.0* starts in less than half a second on a standard PC and takes around 500 Kb of RAM). The Java Virtual Machine is a much heavier process, requiring up to ten times more resources (on the same PC, the standard *Sun JVM* takes 5 seconds to start and 4 Mb of ram).

From this point of view, it's easy to understand why a whole Java Virtual Machine cannot be executed every time a request is made, because too much time would be wasted during each request to spawn and initialize it.

The solution would imply having a virtual machine already up and running when a servlet request arrives. In a server like *JavaSoft Java Web Server*, written in Java, this is not a problem since servlets are executed by the same virtual machine executing the server program itself. This is called “two-tier” model, meaning that the server itself executes the servlet when a client requests it.

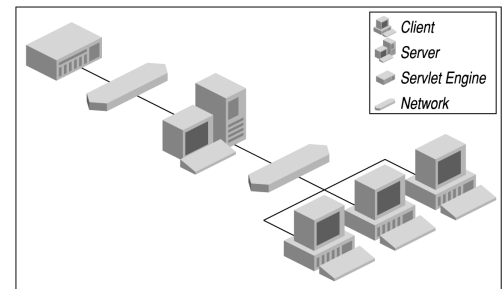


The two-tier model

Including a virtual machine in the Apache Web Server would have created lots of problems: a running JVM would have been included in each Apache process, thus multiplying its “weight” and reducing its performance. This would have

also compromised the stability of Apache itself, since building a Java Virtual Machine is an extremely complicate and difficult task, even more difficult than developing the server itself.

To avoid such problems, Apache JServ was designed following the "three-tier" model. The servlet engine is not part of the web server, but a standalone server application. When executing a servlet request, the web server acts as a client, subrequesting servlets through the network (using the *Apache JServ Protocol* commonly known as the **AJP** protocol), converting the returned data and sending it back to the requesting client.



The three-tier model

Thank to the "three-tier" model Apache JServ can be divided in two main parts: one, included in the web server and acting as an **AJP** client, is `mod_jserv`, and the other one, acting as an **AJP** server, is Apache JServ.

mod_jserv

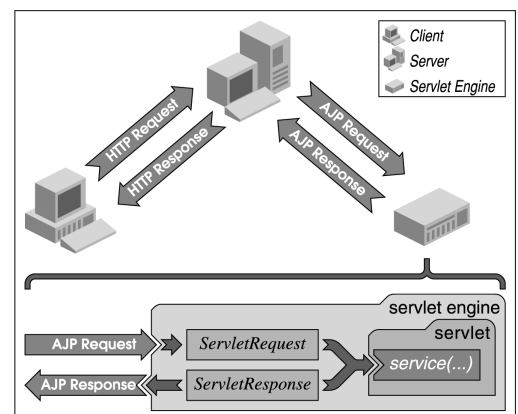
This is a module for the Apache Web Server completely written using C language. Its main task is to forward an **HTTP** request dispatched by the Apache Web Server to the Apache JServ servlet engine using the **AJP** protocol. When the servlet engine processes an **AJP** request, its results are converted back to **HTTP** and sent to the client.

Apache JServ

This is the servlet engine. It's completely written in Java (*100% Pure Java* certification is not available, but all purity tests have been passed). This is a standalone server and for this reason does not require a particular web server to operate (any modular web server could have its own `mod_jserv`). It processes **AJP** requests returning back to its client (the web server) the output of the requested servlet.

To sum up the whole discussion we can draw this figure:

- A client calls the Apache Web Server requesting a servlet.
- Apache handles this **HTTP** request to `mod_jserv` module.
- `mod_jserv` translates the **HTTP** request from the client into an **AJP** requests and contacts Apache JServ servlet engine using a **TCP/IP** network.
- Apache JServ has already performed its initialization process and it's ready to handle **AJP** requests.
- Apache JServ translates the **AJP** request into a **ServletRequest** object, then creates a new **ServletResponse** object used by the servlet to return data to its requester.
- During execution, until the end, all data passed to **ServletResponse** are converted into an **AJP** response and sent back to `mod_jserv`.
- `mod_jserv` translates the **AJP** response into an **HTTP** response and forwards it back to the web server's client.



Apache JServ "three-tier" operation

Apache JServ scalability

Using the “three-tier” model Apache JServ achieves some important and powerful advantages:

Low resource usage of the mod_jserv module

This is very important in both loaded web servers and servers running with little system resources. In fact, since each Apache process has to contain every module, the low memory consumption of mod_jserv allows its use in servers running many different processes without causing troubles in system resources consumption.

Network connection between the two sides

Using **TCP/IP** to connect the two sides of the servlet engine increases the ease of network design and scalability. This is due to the fact that other machines, avoiding sharing the system resources of the web server (memory, CPU, processes), may execute the heavy part of the servlet engine (the Java side).

The Apache JServ servlet engine may be then used in two different modes:

Local mode

This is when both the web server and the servlet engine share the same system resources (are running on the same machine). This is the most common type of operation and it's the fastest one because **AJP** data is not transmitted over the network, but only via the loopback interface. To ease the startup and shutdown of Apache JServ in this mode, mod_jserv is able to spawn automatically the Java Virtual Machine and the servlet engine during web server startup, and to gracefully shutting it down whenever the web server exits. Moreover, the module performs some additional operations like checking the JVM status and re-spawning a new one in case it crashed.

Remote mode

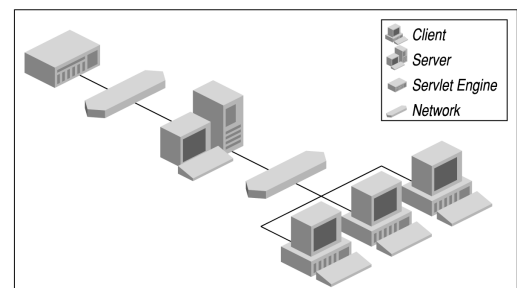
This is when the web server and the servlet engine reside on two different systems. In this mode the module cannot start the servlet engine and control it, but another utility, derived from it, was written: it is called *Standalone Wrapper* and performs the same tasks of startup, shutdown and control that the module does.

This mode may be used to separate the burden of servlet execution from the web server. Another use would be to encapsulate network unaware resources (ODBC databases, serial devices such as modems and fax machines, UPS facilities, etc.) without the need to place a full web server on the remote machine.

Still, on both local and remote mode, the number of web servers and servlet engines placed in a network can lead to different situations:

One to One connection

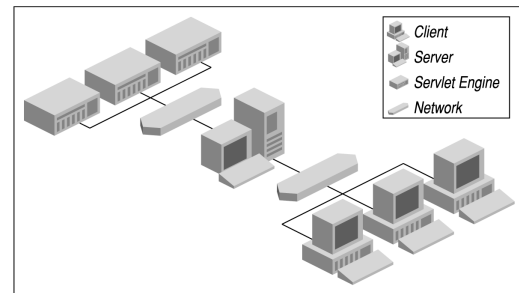
It's the simplest case and it describes a situation where only one web server connects to one servlet engine. This case is by far the most used situation and fits the need of most network environments.



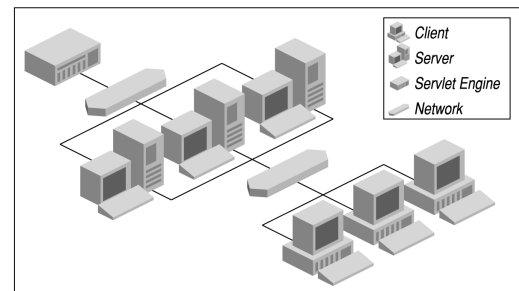
One to One connection scheme

One to Many connection

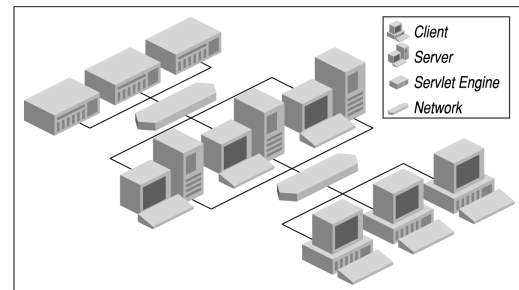
Allows a single web server to connect to several different servlet engines. This is used for web-enabling encapsulation where the resources to encapsulate are fragmented on different machines (an ODBC database running on a host, and a Fax machine connected to another). Another situation would be when resources used by the whole servlet execution are too high for a single machine, and the load needs to be divided between many machines.

*One to Many connection scheme***Many to One connection**

Allows many web servers (or virtual hosts) to connect to a single servlet engine, thus sharing servlets and decreasing network complexity. This could come in help in many cases, for example when a Service Provider wants to deliver servlet through a pool of servers or wants different hosts/virtual-hosts to access to the same servlet engine.

*Many to One connection scheme***Many to Many connection**

Used in very complex network environments, where a pool of web servers connects to a pool of servlet engines.

*Many to Many connection scheme***Security issues**

Security was one of the first concerns in Apache JServ development, in fact using **TCP/IP** for data transmission between the web server and the servlet engine introduces security hazards. The security issue was divided in two parts: the so-called "external security" (preventing intruders to get in the servlet engine, gathering information or destroying data) and "internal security" (preventing servlets running in the servlet engine to make damages).

External security

This was the first issue took in consideration when **AJP** protocol and the servlet engine were designed. Four different kind of attacks were considered:

Intrusion

An intruder is an unauthorized person (or software) that gets into the servlet engine and requests the execution of servlets (if we imagine a servlet executing a free-form **SQL** query to a database, we understand how such servlet could be dangerous from a database system).

To prevent intrusion attacks, an authentication procedure was designed: whenever a client tries to connect to Apache JServ the following steps occur:

- The caller's IP address is checked; if this is not present in Apache JServ configurations, the connection is dropped and the client disconnected. This stage is called IP filtering.
- Apache JServ generates a 4+N bytes (where N means a value specified in configuration files) challenge, built up from the system time (64 bits with 1 ms resolution) and a random number (N bytes long). This challenge key, guaranteed unique per each connection, is then sent to the client.
- The client obtains the challenge data, reads its "secret" key file and concatenates the two buffers. Then it performs the MD5 algorithm over the buffer, obtaining a 128 bits digest string (MD5 message digest algorithm was specified in RFC-1321) and sends it back to the server.
- The servlet engine gets its "secret" key file and performs the same operation. The connection is authenticated and access is granted only if the digest sent by the client and the one produced by the servlet are equal.

Denial of service

Denial of service is the kind of attack aimed to prevent trusted clients to perform requests by overloading the server application.

This kind of attack is usually found in servers that perform heavy operations to execute a client's request. To solve this problems, the IP filtering stage described above comes handy, since Denial of Service attacks range is limited to those hosts with granted access. Since IP filtering is a very fast and simple task, these attacks are reduced by a great amount if they come from not trusted addresses.

Packet sniffing

Packet sniffing means that the attacker looks at the data exchanged in a connection between the servlet engine and one of its clients, gathering information from it. Since protection against this type of attack implies difficult and critical layers, such as global network security (firewalls) or secured protocols (like HTTPS), we decided not to deal with these issues and we designed the AJP protocol without cryptography. In those environments where prevention from these attacks is required, the network must be secured using other means.

Packet spoofing

Packet spoofing is the ability of an intruder to "fake" and re-route packets on a network. This is the most powerful type of attack, but it is the most impractical to perform since very deep security permissions must be obtained in order to be able to do such an attack. AJP authentication and IP filtering do not protect against it. For this reason, like for packet sniffing, the network must be secured in other ways.

Internal security

Many types of servers handle client requests internally and may be visualized as monolithic applications. In these cases, security means protection against external actions aimed to penetrate the barriers. There are servers, though, that relay on third party products (modules or plug-ins) to perform some of the requested tasks. This modularity is a great feature in modern servers, but it adds great security concerns when modules are precompiled and dynamically linkable. In fact, these modules might be used as Trojan horses to penetrate the external security barrier and act undisturbed from the inside.

This is the case in web servers and servlet engines, where both modules and dynamic generators (servlets and CGI executables) may be precompiled and thus hard to analyze for possible security hazards.

The solution is to execute untrusted applications inside a secured environment (sort of locking the Trojan horse in a safe place to limit the range of potential contained intruders). While this has been common practice for CGI (executing the application in a protected security level), very few servlet engines execute servlets using this security model.

Servlet sandboxing

One of Java main goals is complete platform independence. Even if this is one of its great potentials, full portability of compiled code is seriously harmful if not supported by a solid security protection against hostile behavior. For this reason, bytecode verifiers, security managers and execution sandboxes were designed. Today, they are considered secure to the point that Java applets are common practice and do not present security hazards to web browsers.

The Apache JServ servlet engine is designed on this idea, even if, at present, no internal security protection has yet been written. Future releases will complete the implementation following the outlines presented here.

Servlet zones

The servlet engine partitions its execution space into secured, separated zones. These are called *Servlet Zones* and represent a collection of servlets that share the same security restrictions. These zones will act as an abstracted virtual machine from the servlet's point of view, protecting the underlying resources against untrusted access.

It will be impossible for code inside one zone to see or interact with code in another, and for code inside a zone to interact with the servlet engine implementation except through allowed interfaces. Zones can be thought of as solipsistic little servlet engine, but without the overhead of running a separate JVM for each zone.

Current security model

Servlet zones are already present in current Apache JServ implementation (version 1.0b1), even if the security sandbox is not yet written. So, if internal security is an issue today, separate instances of the servlet engine (running with different security levels) may be used on the same machine to separate each zone and protect groups of servlets from one another.

By the most part, this would be needed by an ISP willing to provide a protected servlet environment to its hosted clients, but cannot be used to protect against uploaded code since no restriction are imposed on the executing servlet.

Configuration Examples

Being Apache JServ separated in two parts, the module and the servlet engine, each part store its configurations in different places. `mod_jserv` configurations are placed in the main Apache configuration files (usually `httpd.conf`) while another file (usually `jserv.properties`) contains all the properties required for the servlet engine operation.

Directives (found in `httpd.conf`) follow the following "Apache defined" grammar:

directive_name [**space**] *value* [**space**] *value* [**space**] ...

Properties (found in `jserv.properties`) follow a different grammar, similar to Java properties files:

property_name = *value* , *value* , ...

Each servlet zone requires a specific configuration file to contain specific zone properties. These files use the same grammar of `jserv.properties` file and are usually named using the `zonename.properties` syntax.

Local Mode configuration example

In this example we see a single host, running both Apache and Apache JServ.

We want `mod_jserv` to automatically start the virtual machine and Apache JServ, so in `httpd.conf` we place the `ApJServManual Off` directive and we specify the location of `jserv.properties` file through `ApJServProperties`.

In `jserv.properties` file `mod_jserv` finds all the properties required to start the virtual machine: `wrapper.bin` specifies our virtual machine's binary, `wrapper.path` the path environmental variable to use and `wrapper.classpath` the classpath variable.

If path and classpath are not specified, these are inherited from `mod_jserv` parent process.

At initialization Apache JServ parses the `jserv.properties` file, and there it finds all its configuration properties: `port` is the port used for connections, `zones` is the list of all zone names, `zonename.properties` are per-zone configuration files, `security.allowedAddresses` is the list of hosts enabled to connect to the engine and `security.SecretKey` is the shared secret key used for client authentication. Obviously the shared secret must be known also by `mod_jserv` and so in `httpd.conf` the `ApJServSecretKey` directive is specified; in this case Apache JServ and `mod_jserv` use the same secret file, but they could also use two exact copies of it.

Two servlet zones are defined: `weird` and `strange` (their per-zone configuration files are, respectively, `weird.properties` and `strange.properties`). These two zones share one same repository `/usr/java/jsp.jar`, and they both have another repository: `/usr/java/servlets/weird` and `/usr/java/servlets/strange`.

`mod_jserv` wants to see the `weird` zone under `/servlets` and the `strange` zone under `/strangeservlets`. To do that in `httpd.conf` a default protocol to use (`ApJServDefaultProtocol`), a default host to connect to (`ApJServDefaultHost`) and a default port (`ApJServDefaultPort`) are defined. Then to activate the servlet-zone/URI-path aliasing it the `ApJServMount` directive is used.

httpd.conf:

```
ApJServManual Off
ApJServProperties /usr/java/servlets/jserv.properties
ApJServDefaultProtocol ajp11
ApJServDefaultHost localhost
ApJServDefaultPort 8007
ApJServMount /servlets /weird
ApJServMount /strangeservlets /strange
ApJServLogFile logs/jserv.log
ApJServSecretKey /usr/java/servlets/jserv.secret.key
```

jserv.properties:

```
wrapper.bin=/usr/java/jdk/bin/java
wrapper.classpath=/usr/java/classes,/usr/java/classes.jar
wrapper.classpath=/usr/bin,/bin,/usr/java/bin
port=8007
zones=weird,strange
weird.properties=/usr/java/servlets/weird.properties
strange.properties=/usr/java/servlets/strange.properties
security.allowedAddresses=127.0.0.1
security.SecretKey=/usr/java/servlets/jserv.secret.key
```

weird.properties:

```
repositories=/usr/java/servlets/weird,/usr/java/jsp.jar
```

strange.properties:

```
repositories=/usr/java/servlets/strange,/usr/java/jsp.jar
```

Remote Mode configuration example

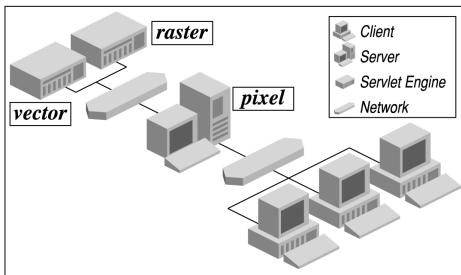
In this example we have three hosts: *pixel* running Apache web server, *vector* and *raster* running Apache JServ: *vector* has two servlet zones, *rain* and *snow*, while *raster* just one: *fog*.

The web server *pixel* wants to deliver *vector*'s servlets under **rain**servlets and **snow**servlets URI paths, and *raster*'s under **winfog**. *vector* is a Sun Solaris host, running Sun's Java Virtual Machine (the Java binary is `/usr/java/jdk/bin/java`). It allows accesses only from *pixel* and the repositories for rain and snow are respectively found under `/usr/java/jserv/rain` (a directory) and `/usr/java/jserv/snow.jar` (a jar file).

It also finds its secret key under `/etc/jserv.secret.key`.

Raster is a Microsoft Windows 95 machine. It uses the Windows VM (`c:\windows\jview.exe`) and its zone repository is again a jar file: `c:\windows\java\servlets.jar`. Differently from *vector*, *raster* allows access also to itself, through its loopback interface (see in `jserv.properties` file the **security.allowedAddresses** property).

To "mount" the three different servlet zones *pixel* defines the default protocol, host, port and secret key file used by all **ApJServMount** directives. When "mounting" *raster*'s **fog** zone then overrides both the host and the secret key, specifying their values in the **ApJServMount** directive.



Remote Mode example network layout

pixel's httpd.conf:

```
ApJServManual On
ApJServDefaultProtocol ajpvl1
ApJServDefaultHost vector
ApJServDefaultPort 8007
ApJServSecretKey conf/vector.secret
ApJServMount /rain/servlets /rain
ApJServMount /snow/servlets /snow
ApJServMount /winfog raster/fog conf/snow.secret
ApJServLogFile logs/jserv.log
```

vector's jserv.properties:

```
wrapper.bin=/usr/java/jdk/bin/java
wrapper.classpath=/usr/java/classes
port=8007
zones=rain,snow
rain.properties=/usr/java/jserv/rain.properties
snow.properties=/usr/java/jserv/snow.properties
security.allowedAddresses=pixel
security.SecretKey=/etc/jserv.secret.key
```

vector's rain.properties:

```
repositories=/usr/java/servlets/rain
```

vector's snow.properties:

```
repositories=/usr/java/servlets/snow.jar
```

raster's jserv.properties:

```
wrapper.bin=c:\windows\jview.exe
wrapper.classpath=c:\windows\java\classes.jar
port=8007
zones=fog
fog.properties=c:\windows\java\fog.properties
security.allowedAddresses=raster,localhost
security.SecretKey=c:\windows\java\secret.key
```

raster's snow.properties:

```
repositories=c:\windows\java\servlets.jar
```