The WebSphere Application Server architecture and programming model

by E. Bayeh

This paper discusses the infrastructure that IBM is providing to support the World Wide Web and to facilitate Web applications and commerce. This effort started as an architecture called the Network Computing Framework (NCF), and is now the foundation of the IBM WebSphere Application Server™. The WebSphere Application Server is a product IBM first delivered in June 1998. In this paper we discuss this architecture and programming model. We start with a brief introduction and history of the NCF, then examine the architecture of the WebSphere Application Server. We take a close look at the core run time of the WebSphere Application Server, then delve into the Java™ programming model that supports this architecture. We also present the reasons why Java is a prominent part of this architecture, and see what relevant technologies Java provides for this run time.

The World Wide Web grew from a small network to facilitate document sharing among a handful of physicists to a large network of networks, connecting thousands, perhaps millions, of computers and computer users together. The most amazing fact is that all this growth occurred in a very short period of a few years. Today, the World Wide Web is not only used to exchange documents, but also to run business operations and handle finances. It is both a huge repository of information and a marketplace for commercial and financial transactions.

The Java** programming language is taking on a new role on the World Wide Web. Having started mainly as a way of writing portable client applications (applets) that run on Web browsers, it is now being used to write stand-alone applications, as well as server-side applications (servlets) that run on Web servers. In a later section we discuss the reason for deploying Java on servers and stand-alone applications, and the possible advantages of doing so.

The paradigm shift on the Web has been to change the role of Web (HTTP—HyperText Transfer Protocol) servers, from traditionally serving flat HTML or HyperText Markup Language (static) files and running a few programs and gateways using the CGI-BIN interface (Common Gateway Interface as set up in a UNIX** directory), to hosting Web applications, serving live (dynamic) content. It has evolved from a basic static publishing model to an interactive application model. In a sense, Web servers are now application servers. Client/server computing has a whole new life. Web browsers, the client side of this paradigm, are low-cost and pervasive. They are excellent as an application client, because they are widely adopted (almost every computer user has access to a browser) and allow the dynamic download of application code from the server (Java applets). Browsers also provide peace of mind to the user because of their built-in security model. The client application (applet) cannot introduce a virus, is discarded after use, and can be assumed to be trusted,

0018-8670/98/$5.00 © 1998 IBM

if it is digitally "signed" by a trusted source. Deployment of client/server applications in this paradigm is very easy and cost-effective. The server application (servlet) and the client part (applet) are hosted on a "Java-compliant" Web server, and users are simply pointed to a uniform resource locator (URL).

The future is even more promising now that Internet Inter-Orb Protocol (IIOP), an object-oriented (CORBA**—Common Object Request Broker Architecture) communications protocol, is being increasingly supported by Web servers and clients. HTTP is a nonconversational protocol and is not really designed to be used as a client/server protocol. IIOP solves many of the current problems with HTTP and makes applet-to-servlet communications more efficient.

Even Web applications (applications designed for use on the Web) are undergoing a metamorphosis. The transition to dynamic content was initially done through CGI programming and HTML forms. Every server supported CGI-BIN, and every browser supported HTML forms, making this an easy choice for developers. HTML forms allowed a browser user to enter some data on a form and then post that form to the Web server, which started a CGI program that usually was a gateway to a third-tier application such as a database. The CGI program fetched data from the third tier, formatted the data into HTML, and sent the data back to the client. Although this model was successful, and most Web applications and gateways today are still deployed that way, there are limitations. CGI programs are slow and not portable. HTML forms do not have the user interface (UI) that most sophisticated users are used to. This model was great for doing a simple database query and viewing the results. But what about a banking application using forms and CGI programs? The answer has been, in some cases, to move the client-side UI into Java.

Today almost every browser supports Java applets and, more importantly, is moving the server-side of the application to higher-function and higher-performance Java servlets. Even newer technologies on the horizon are providing more sophisticated function, namely, the newly announced Enterprise JavaBeans** (EJBs) and the Internet Inter-Orb Protocol (IIOP). EJBs form a server-side component architecture that provides a transactional, persistent object programming model, and IIOP is a protocol for object communication that is more efficient and functional than simple HTTP. IIOP allows objects to be distributed and encapsulates the physical location of the objects from the application. IIOP also allows the interoperability between all CORBA-compliant objects and sets the stage for true distributed object programming. Note, however, that when this paper was written, the use of IIOP and EJBs on the Web was not yet prevalent. Their use has only just begun, but before the end of 1998, IBM will provide the framework necessary to start building and deploying EJB applications.

The WebSphere Application Server architecture

The IBM WebSphere Application Server* is the result of the evolution of what was traditionally the "Web" server. The addition of "application" acknowledges the fact that this server is no longer simply serving HTML but also industry-strength business applications. In some ways it is also the gateway to data and applications on back-end, third-tier systems. A large number of applications on the Web server are simply gateways to an existing back-end application or server and use a set of "connectors" for access to this back end. Figure 1 shows the three tiers of the WebSphere Application Server: the HTTP engine, the servlet engine, and the Enterprise JavaBeans engine (enterprise bean container). The server is designed to be open in order to work with general industry tools. The WebSphere Application Server as illustrated in Figure 1 is an implementation of IBM's Enterprise Server for Java specification. For more information on Enterprise Server for Java, see Brackenbury et al.[1]

The *core HTTP engine* depicted in Figure 1 handles HTTP Web requests: requests for static resources such as GIF (Graphic Interchange Format) files, HTML files, etc.; requests for CGI programs; and requests for plug-in applications. Servlet requests are passed on to the Java application engine (discussed below), after undergoing the normal Web server authentication, authorization, and logging steps. This tier is the first line of scalability and the easiest tier to scale, since most requests handled here are static and noninteractive. Although static requests are normally short-lived, the core engine could quickly get overwhelmed with hundreds or thousands of concurrent requests. Scaling the core engine for static content is simple; the prevalent solution is to use an HTTP "sprayer" (a load balancer), where multiple servers are run in a cluster, sharing the same static resources via a shared file system. The most critical elements of this engine are response time and throughput. *Response time* is the turnaround time of handling a sin-

Figure 1 Logical tiers of the application server

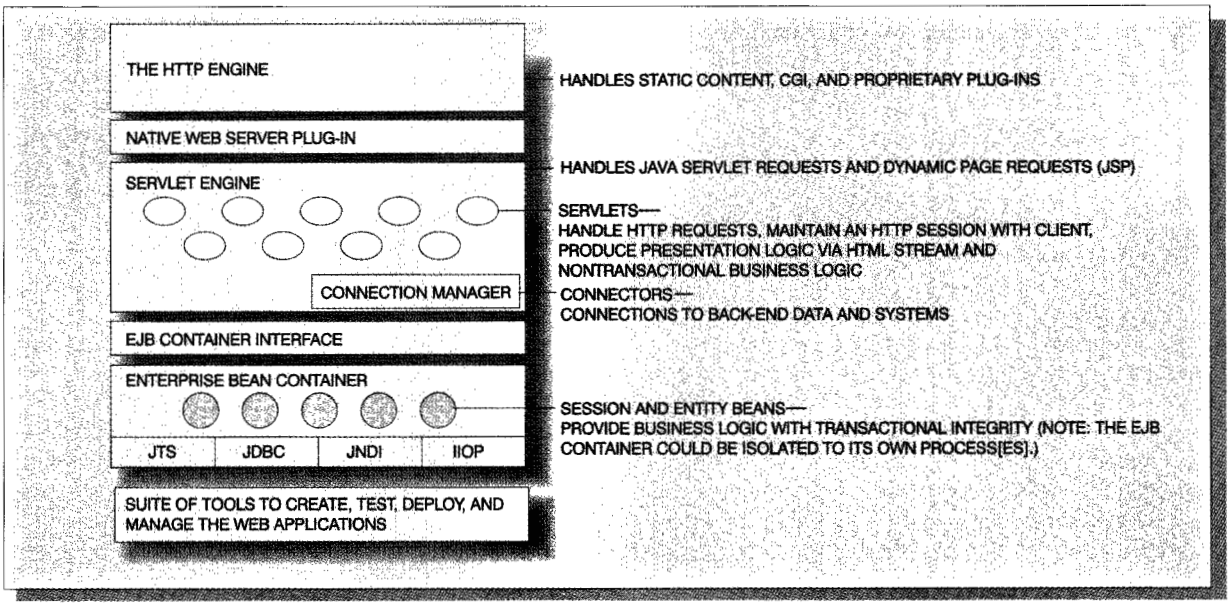
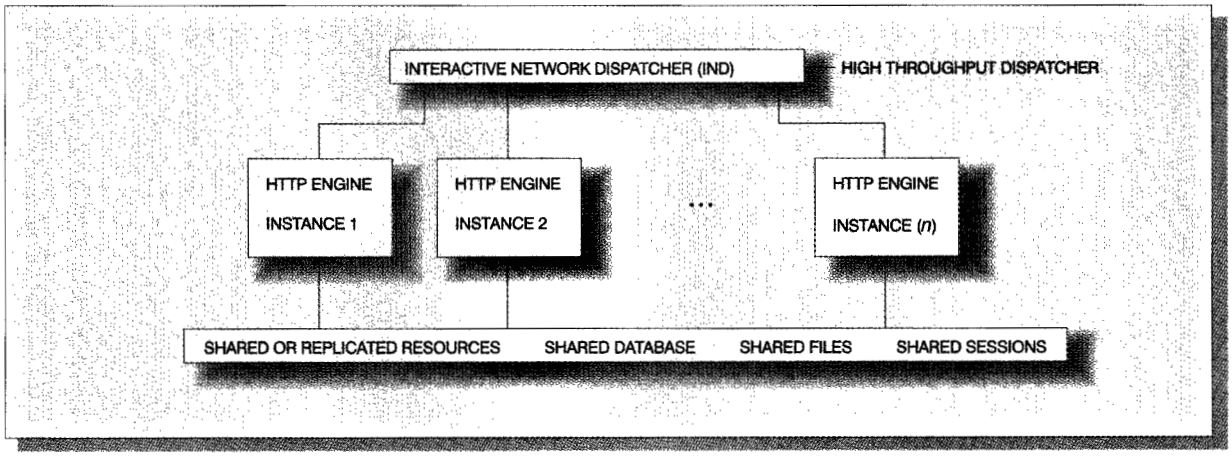


Figure 2 Clustering the HTTP server subsystem



gle (simple) HTTP request; *throughput* is the total number of these requests that the engine can handle, typically in one second. The engine is optimized to meet these criteria, and where a single engine fails the throughput demands of a Web site, multiple HTTP engines are used concurrently behind a sprayer called the Interactive Network Dispatcher (IND) as shown in Figure 2. IND, an IBM product, is a front end to a Web site that balances load and distributes HTTP requests to a number of HTTP servers (typically via a round-robin method, but could also do intelligent routing based on workloads).

Figure 3 Servlet engine run-time components

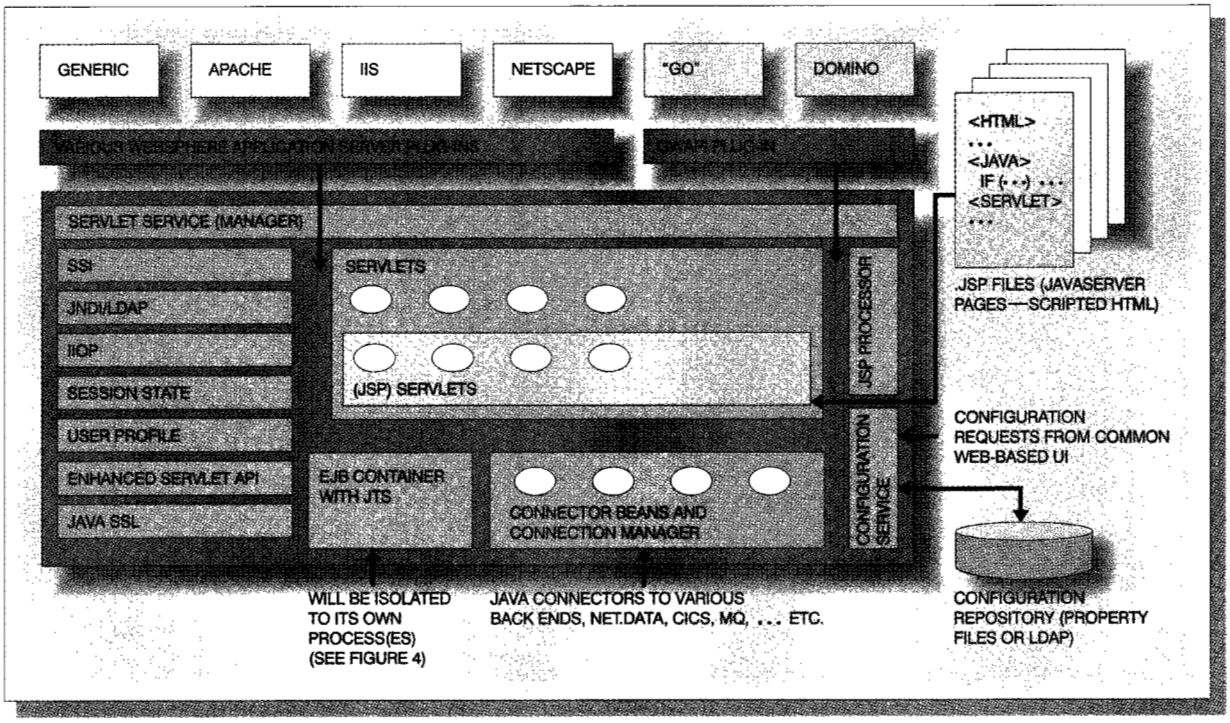


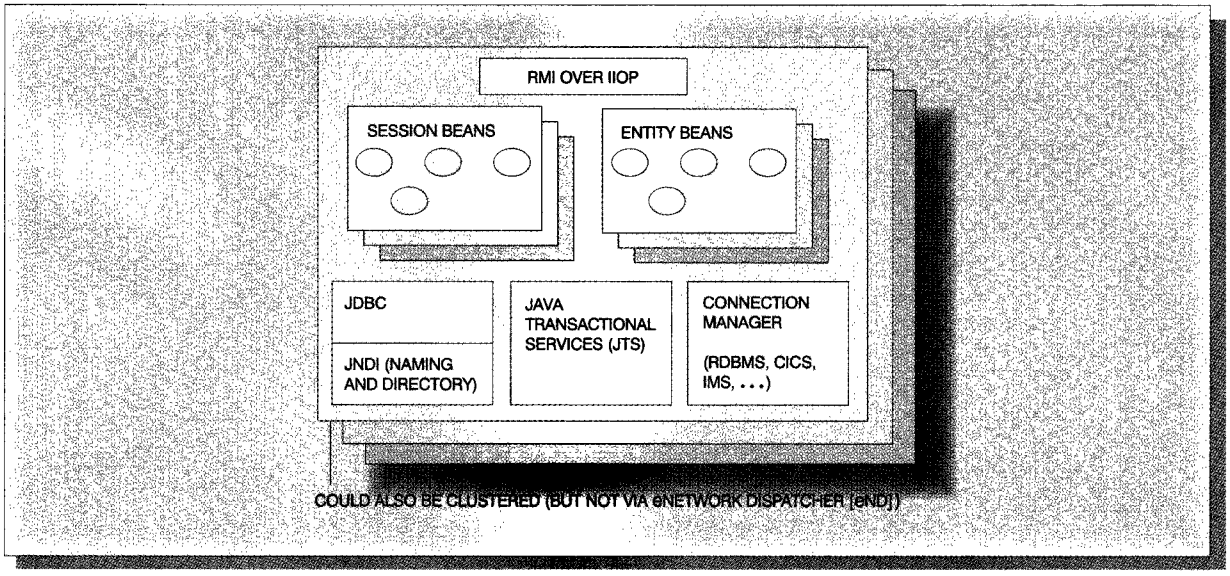
Figure 3 shows the run-time architecture of the *Java application* (*servlet*) *engine*. The run-time environment, shown as the sets of rectangles in blue-green, plugs into major Web (HTTP) servers via proprietary plug-in application programming interfaces (APIs) such as Internet Server API (ISAPI) and Netscape Server API (NSAPI), and then routes servlet requests to the servlet manager which then takes care of handling the request and passing the data back to the client. This engine also handles requests for JavaServer** pages (JSP) or server-side HTML scripting. These dynamic requests are not normally as short-lived as the static ones discussed previously. Therefore, the limit of how many concurrent requests this engine can handle is substantially smaller than the above, and the response time is longer. This is to be expected, since we are now running *applications* and dynamic content, and not just sending static bytes back to the client. Making this tier an "industry-strength" solution requires the multiprocess support discussed later (seen later in Figure 5).

Figure 4 shows the architecture of the *Enterprise JavaBeans engine* run time. This engine handles running the business logic, ensuring transactional integrity. It is basically a managed object framework and deals with life-cycle and persistence issues of the enterprise beans it manages. It has identical scalability issues as the servlet engine, perhaps compounded further by the fact that this engine has to deal with issues typical of a transaction monitor, and it also accepts IIOP connections.

Servlet queues. To overcome the limitation of running all servlets and Web applications in a single Java process, a new servlet engine architecture was developed to allow multiple Java processes to be run on the same machine. Requests coming to the Web server are routed (based on policy) to one or more queues, each serviced with one or more Java processes. The benefits of having multiple processes are discussed in the next subsection. The administrator defines how many queues to set up (the default is one), and then defines a *policy* associated with that queue.

The *queue policy* defines the URLs that are serviced by the queue, the number of processes servicing the

Figure 4 The enterprise JavaServer subsystem



queue (the default is one), and the Java and security environment of the queue processes. Examples of the Java environment that can be configured via policy are CLASSPATH, JVM (Java Virtual Machine) path, and security.

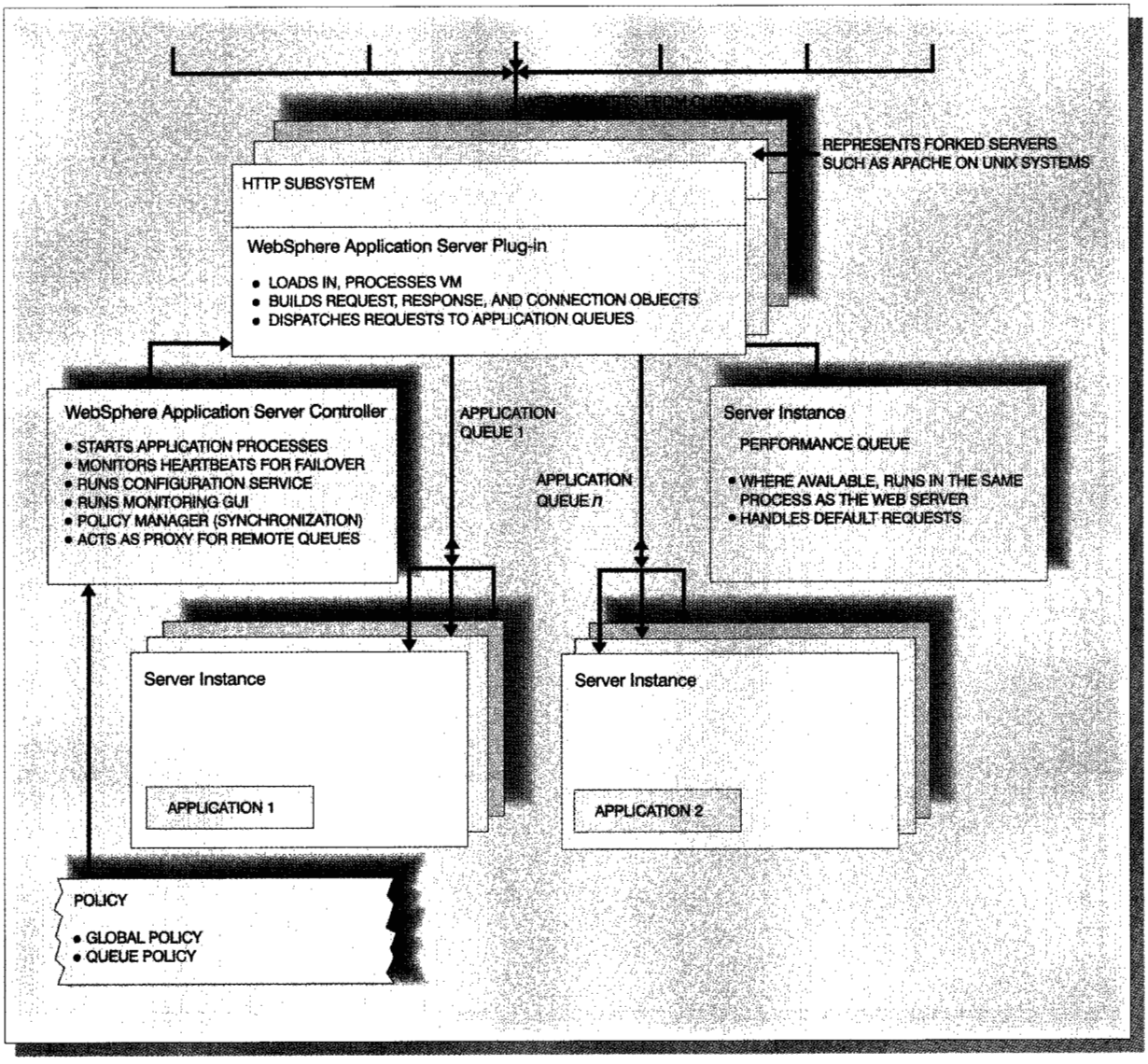
In Figure 5, except for the policy information defined earlier, all of the WebSphere Application Server instances are basically the same. For simplicity, they all share the same configuration files.

Benefits of servlet queues. As was stated, multiple process support is essential for a serious business server. Even the low-end server requires a certain amount of reliability, security, and throughput. The most important goal of the multiprocess architecture is to maintain simplicity and only provide support when needed, without sacrificing single process efficiency and simplicity. The policy is designed to allow running in a single process (with no queues) and to easily add queues if needed. Following are the benefits of having multiple Java processes.

- Availability and reliability 24 hours a day, seven days a week—One of the obvious drawbacks of running a single-server process occurs when the process crashes, hangs, or even temporarily stops. As a result, the server is basically gone. Java has not yet matured to the "crash-proof" level required for serious, heavy-duty application servers. Running multiple processes provides a way for uninterruptible operation of the server, even if a process or an application occasionally crashes. Queues can be configured for fail-over by specifying more than one clone in the queue policy. This action allows other processes to take over work when one of the processes is not accepting any.

- Load balancing, multiprocessor server utilization, and performance—This group of benefits is only achieved if a single-server process does not completely utilize all of the available CPU. There is obviously no benefit to load balancing if a single process overloads the CPU of the server. If the Java process is not CPU-bound, there is a limit of classes and requests that it can handle before bottlenecks and thread deadlocks start causing performance degradation. Load balancing across multiple processes alleviates the occurrence of these performance degradations and exploits multiprocessor server hardware architecture. There is also the notorious garbage collection (GC) problem, which occurs when the Java Virtual Machine chooses to carry on housekeeping tasks such as GC, and the entire virtual machine comes to a halt. Spreading GC to multiple processes also alleviates this prob-

Figure 5 Application queues and multiple process support



lem, assuming that not all the processes perform housekeeping tasks at the same time (very unlikely).

- Bottlenecking the HTTP server—The HTTP engine is typically structured to have a finite number of worker threads that process Web requests. For a Java request, the HTTP worker thread typically passes the request to the WebSphere Application Server engine and waits for the request to finish before returning to service a new request. Since Java requests take longer than an average static Web request, the throughput of the Web server is reduced when the amount of Java requests it services is increased. A simple benchmark shows that on Windows NT**, the Web server can handle 600 static requests per second compared to 100 Java requests per second. The multiprocess architecture includes a feature called connection passing. This feature is not available on all Web servers, but

when it is available, the HTTP thread will put the request on the application queue and return (not wait) to service new requests. The Java application process will pull the request off the queue, pick up the connection left open by the HTTP thread, and take over that connection. This action requires the Java process to then perform the HTTP functions that the HTTP thread would have done, but this is a small price to pay for improving the throughput of the HTTP engine, while running Java application requests.

- Application confinement or isolation—Java today allows applications running in the same virtual machine to intentionally, or unintentionally, interfere with one another. One can argue that these applications should be trusted, since they were adequately tested by the developers to ensure that there are no problems, but that is not enough for certain mission-critical applications. An administrator, for security reasons, can choose to confine an NCF application (one or more servlets) to its own virtual machine, thus preventing the possibility of other servlets interfering with it.

- Pluggability of JVMs—This architecture allows a particular Java environment (including a virtual machine) to be defined per application queue. This action allows a particular site to run a mix of virtual machines at the same time. Although this action is not typically recommended because of bugs and different Java implementations developed by vendors, there are cases where it will be very desirable and even required.

- Debuggability—Debugging servlets is currently a very difficult task, because the servlets are extensions of the Web servers and usually need to be debugged in the same environment in which they run. There are many debugging techniques; the technique being referred to here is the use of remote debuggers. Remote debuggers require that the virtual machine in which the application is running be started in a specific mode. The multiprocess architecture allows a particular queue to be made debuggable, without affecting the entire site.

Use of Java on the server

Until Java Development Kit Version 1.1 (JDK** 1.1) became available, Java did not have the reliability and performance required by heavy-duty server applications. Thus the use of Java on servers is new and not pervasive. With Version 1.1, the reliability, functionality, and performance of Java has improved and actually shows promise of exceeding C++. Java offers the standards discussed in this section to make it a powerful candidate for NCF applications. The building blocks of a server-side NCF application are servlets, JSP files for scripted, dynamic HTML, and session state for maintaining HTTP session information. JavaBeans**, and more recently, Enterprise JavaBeans, are the component architecture for building Java applications.

JavaBeans. JavaBeans is an architecture that allows the development and utilization of reusable software components in Java. The application programming interfaces (APIs) of JavaBeans were designed to be simple, but powerful. The "official" JavaSoft definition of a JavaBean is the following: "A JavaBean is a reusable software component that can be manipulated visually in a builder tool."[2]

This definition is not exactly accurate, as we discuss later. Beans can also be manipulated by the servlet engine, which is not a builder tool and is certainly not visual. However, this definition serves to indicate that there is a *contract* between the JavaBean and the tool, and this contract is the JavaBeans API. The consumer of the beans (visual builder tool, or server run time) uses this contract to understand what the bean does and how to set and retrieve its properties or invoke its methods. The important features of a JavaBean is its support for the following:

- Introspection—how the consumer of the bean can analyze and examine how the bean works
- Customization and properties—used to customize the appearance and behavior of the bean
- Events—a simple communication metaphor used to connect beans together
- Persistence—allows customized state of the beans (consisting of its properties and local variables) to be saved and reloaded later

Servlets. Servlets are Java programs that run on a server. The basic form of servlets, called GenericServlets, run on any generic server. To handle HTTP requests, a subclass of servlets called HttpServlet is available. The HttpServlet class is an abstract class that simplifies writing HTTP servlets. It extends the GenericServlet base class and provides a protocol-handling framework. Because it is abstract, servlet writers must subclass it and override at least one method. The methods now described are normally overridden.

The *init( )* method is called when the servlet engine first loads the servlet. It only executes once and is not repeated throughout the life of the servlet instance. A servlet is not required to override the init( ) method; the default provided is usually adequate. Typical things that could be done in this method are to initialize database connections and load default data.

The *destroy( )* method is called when the engine is getting ready to unload a servlet. Like the init( ) method, it is also only called once. The default destroy method is usually also adequate. Typical things to do in the method are housecleaning operations such as releasing used resources and closing active connections.

The *service( )* method is called for every client request. It is the heart of the servlet. It is called by the Web server and is passed a request and response object. When the server calls the HttpServlet service( ) method, it determines whether the request is a GET, PUT, or POST, and calls the appropriate doGet, doPut, and doPost methods accordingly. Uses of the methods are as follows:

- *doGet*, for a servlet to be invoked by an HTTP GET. Where practical, the *getLastModified* method should also be overridden to facilitate caching the HTTP response data.
- *doPost*, for a servlet to be invoked by an HTTP POST request
- *doPut*, for a servlet to be invoked by an HTTP PUT request
- The life-cycle methods *init* and *destroy*, if the servlet writer needs to manage costly resources that are held for the lifetime of the servlet. Servlets that do not manage such resources do not need to specialize these methods.
- *getServletInfo*, to provide descriptive information through the administrative interfaces of a service

The HttpServlet class also provides methods for the HTTP 1.1 extensions: OPTIONS, DELETE, and TRACE.

By subclassing HttpServlet and implementing the doGet method, a servlet automatically supports the GET, HEAD, and conditional GET operations of HTTP. Adding support for the getLastModified method enables caching, thus improving Web server performance. Servlets are typically *singletons*, meaning that a single instance of the servlet is created to handle multiple client requests. Since these requests could be concurrent, servlets must be written to handle simultaneous requests and multithreading. Access to shared resources such as class variables and in-memory data must be synchronized, or the servlet is not "thread safe" and may not run properly.

When the HTTP subsystem calls the service( ) method of a servlet, it passes what is commonly called the request and response objects as parameters. The objects HttpServletRequest and HttpServletResponse are the way in which the servlet communicates with the server, and ultimately with the client. The methods of the request object obtain information about the client environment, the server environment, and any information provided by the client (for example, form information set by GET or POST). The methods used to retrieve this information are: getParameterNames( ), getParameter( ), getParameter-Values( ), and getQueryString( ).

The servlet invokes the methods of the response object to send the response that it has prepared back to the client. Its methods allow the response header and the response body to be set. The response object also has the getOutputStream( ) method to return a ServletOutputStream object. The print( ) and println( ) methods of the ServletOutputStream object are used to write the servlet response back to the client.

Among many ways to invoke a servlet are the following:

1. The most typical way is to use a Web browser to open the servlet either by codename, for example, http://www.webserver.com/com.ibm.myapp.myservlet.class, or by using a defined alias for the servlet, for example, http://www.webserver.com/myApp, assuming that the administrator has defined the alias "myApp" as an instance of the servlet "com.ibm.myapp.myservlet.class."
2. In HTML forms, the servlet on the ACTION attribute of the <FORM> tag in an HTML file can be specified.
3. In server-side includes, the servlet within the <SERVLET> tag in an SHTML (Server-side HTML) file can be specified.
4. In JavaServer pages (discussed later), the Java coding for a servlet can be embedded or the <BEAN> tag can be used to embed a JavaBeans servlet.
5. A more advanced feature is servlet chaining and filtering. A servlet filter or a servlet chain can be specified.[2]

Servlet beans. Servlets that are themselves also JavaBeans are referred to as "servlet beans." There are two distinct advantages to making servlets beans. The persistent state of a bean, and its configuration information, can be stored in a serialized file. The configuration can be updated dynamically and takes effect immediately. By default, the servlet engine assumes that every servlet is a bean. Properties are the initial arguments that are passed to the servlet bean. There are three ways to install a servlet bean into the servlet engine:

1. By using the class (myServlet.class) of the servlet bean
2. By using a serialized instance of the servlet (myServlet.ser). This can be thought of as one instance of the servlet.

Servlets provide the infrastructure to maintain session data across multiple client requests and multiple servlets.

3. By providing a JAR (Java archive) file (myServlet.jar) with the class files or the serialized instance file, or with both

These ways could be placed in the *servlet* directory, the *servlet beans* directory, or any directory in the *CLASSPATH*. However, automatic reloading of servlets (if they change) does not occur for servlets that reside in the CLASSPATH.

Maintaining state and session information. To overcome the limitation of the "nonconversational" HTTP, servlets provide the infrastructure to maintain session data across multiple client requests and multiple servlets. The APIs provided allow a servlet to share user and application data across multiple client requests and also across multiple servlet instances. This infrastructure is called *session tracking*. Session tracking gives servlets the ability to check on the status of a user as the user moves through the site. It is a flexible, lightweight mechanism that serves as a basis for more sophisticated state models, such as personalization and persistent user profiles, and servlets can use this facility to track who is doing what on the site. The official Sun Microsystems definition of a session is: "A session is a series of requests from the same user that occur during a time period."[2]

The servlet engine maintains a user state by creating a session object for each user that has used the site. These session objects are stored and maintained on the server for a defined length of time, after which they are discarded (although there is a mechanism to make them persistent). The user is assigned a new session object and a unique session identifier when first making a request to a site. The session identifier matches the user with the session object in subsequent requests. The session object is then passed as part of the request to all servlets that handle the request. Servlets can then add information to session objects or read information from them.

Following is the list of APIs, as defined in the Java interface "javax.servlet.http.HttpSession":

- *getId( )*—Returns the identifier assigned to this session. Different kinds of sessions use different identifiers, such as byte arrays, strings, and multicast network addresses.
- *getSessionContext( )*—Returns the context in which this session is bound.
- *getCreationTime( )*—Returns the time at which this session representation was created, in milliseconds since the epoch. This may not correspond directly to the time the session itself was created, since that may not be reliably knowable for long-lived multiparty sessions.
- *getLastAccessedTime( )*—Returns the last time this session representation was accessed by the session-level infrastructure, in milliseconds since the epoch. Access indicates a session protocol level access to the session, such as a new member joining, an existing member leaving, a "keep alive" being sent, or a new connection being established using session data. Application-level operations, such as getting or setting a value associated with the session, are not reflected in this access time. This information is particularly useful in session management policies. For example, a session manager could leave all sessions that have not been used in a long time in a given context, or the sessions might be sorted according to age to optimize some task.
- *invalidate( )*—Causes this representation of the session to be invalidated and removed from its context.
- *putValue( )*—Binds the specified object into the application layer data of the session with the given

name. Any existing binding with the same name is replaced. If the new (or existing) value implements the SessionBindingListener interface, it is notified appropriately.
- *getValue( )*—Returns the object bound to the given name in the application layer data of the session. Returns null if there is no such binding.
- *removeValue( )*—Removes the object bound to the given name in the application layer data of the session. Does nothing if there is no object bound to the given name. If the value implements the SessionBindingListener interface, it is notified appropriately.
- *getValueNames( )*—Returns an array of the names of all the application layer data objects bound into the session. For example, if the request is to delete all of the data objects bound into the session, this method is used to obtain their names.
- *isNew( )*—Considers a session to be new if it has been created by the server, but the client has not yet acknowledged joining the session. For example, if the server supported only cookie-based sessions and the client had completely disabled the use of cookies, then calls to HttpServletRequest.getSession( ) would always return new sessions.

JavaServer pages. JavaServer pages (JSP) provides for a powerful scripting solution that allows dynamic HTML generation on the server side. JavaServer pages allows the Web developer to do simple scripting on the server side that takes advantage of JavaBeans and provides a powerful dynamic content generation facility on the server side, separation of dynamic content generation, and the presentation of the content.

JSP allows the clear separation of presentation logic (HTML) from the application logic (programmatic). Web designers can design the presentation layout of their pages using any one of the numerous HTML creation tools available, and then the dynamic Web logic is added with minimal programming.

JavaServer pages permit a scripting language to be embedded in Web pages (HTML documents). Before the page is served, the JavaServer pages syntax is parsed and processed into a servlet on the server side. The servlet so generated will output real dynamic HTML content back to the client. The scripted HTML file will have a .jsp extension to identify it as a JavaServer pages file to the server.

JavaServer pages can be invoked in two ways:

- Directly from the client—A request comes into a JavaServer page. The page accesses reusable JavaBeans components that perform particular well-defined computations (such as accessing a database) and store result sets as bean properties. The page uses such beans to generate dynamic content and presents it to the client. The contract that the JavaServer pages developer cares about is the bean interface.
- From another servlet—A request comes in to a servlet that generates the dynamic content that the response would contain. The servlet invokes a JavaServer page that will present the content generated from the servlet.

The following subsections provide a look at the JSP syntax, taken from the JSP specification developed jointly by IBM and JavaSoft.

JSP directives. Directives are declarative statements that specify such things as the scripting language being used and the class and interfaces a servlet extends or implements. The general syntax of the JavaServer pages directive is:

<%@{*variable* = "<value>"} + %>

where *variable* is one of the five types described below.

The *language* variable defines the scripting language used in the file. The scope of this tag spans the entire file. When used more than once, only the first tag is significant. If omitted entirely, the default scripting language used is JavaScript** (for the Java programming language). This value is the only one that the language variable can accept at this time.

The *method* variable defines the name of the method that will contain the body of generated code from the script. By default, the method defined in the generated servlet is service. When a specific method name is used, the generated code becomes the body of the specified method name.

The *import* variable defines the list of packages that will be imported by the servlet. This list consists of comma-separated Java language package names or class names that the servlet imports.

The *implements* variable defines the list of interfaces that the generated servlets implement. The value for this variable is a comma-separated list of Java language interface names.

The *extends* variable defines the super class of the generated servlet. The value is the name of the Java language class from which the servlet extends.

Sample directive: <%@*method* = "doGet"%>

JSP declarations. Declarations define class-wide variables for the servlet class and are defined within a SCRIPT tag.

```
<SCRIPT runat=server >
. . .
</SCRIPT>
```

Sample declaration:

```
<SCRIPT runat=server>
int i = 0;
String foo="Hello";
</SCRIPT>
```

JSP scriptlets. The body of the scriptlet is the heart of the body of the method (doGet, doPost, etc.) of the generated servlet. The script can rely upon a set of predefined variables. These variables are: *request* (the request object as defined by javax.servlet. ServletRequest), *response* (the response object as defined by javax.servlet.ServletRequest), *out* (the servlet output writer class as defined by java.io.PrintWriter), and *in* (the servlet input reader class as defined by java.io.BufferedReader).

The code itself is embedded between <% and %> tags. For example:

```
<%
 response.getPrintWriter( ).print("Hello");
 %>
<%
foo = request.getParameter("Name");
out.println(foo);
%>
```

JSP expressions. JSP expressions is syntax that defines an expression that will be evaluated. The value of the expression will be substituted in place of where the expression occurs. For example,

```
<% = foobar %>
```

will substitute the value of foobar in place of the tag.

JSP beans. One of the most powerful features of JavaServer pages is that JavaBeans, and soon Enterprise JavaBeans, can be accessed from within a JavaServer pages file. Any of the following actions can be performed on the bean. The bean can be: created from a serialized file or a class file, referred to from an HTTP session, or passed to the page from a servlet.

The syntax for the bean tag is:

```
<BEAN name=" <value>"
  varname=" <value>"
  class=" <name> " introspect="{yes|no}"
serializedfile=" <value>" create="{yes|no}"
  scope="{request|session}">
</BEAN> (The close tag is optional.)
```

JDBC. Although JDBC** is not an acronym, it is generally thought to mean Java Database Connectivity. It is a set of APIs, modeled after Open Database Connectivity (ODBC), that provide Java programs with the ability to access all relational databases and execute Structured Query Language (SQL) statements. It is a way for Java applications, and more specifically in this case, servlets, to talk to a variety of different databases without being dependent on a specific vendor. Taking this a step further, IBM provides a "JDBC connection manager" as part of the servlet run time. This connection manager provides a beans interface and maintains a pool of active JDBC connections to a database, hence improving performance and throughput.

Enterprise JavaBeans. Enterprise JavaBeans (EJBs) take Java to the next level. They provide a component architecture for multitier, distributed Java applications. In one sense, EJBs extend normal JavaBeans components to support server components. The EJB infrastructure provides transactional and system services for the application components, making distributed, client/server applications easier than

ever to develop, deploy, manage, and maintain. The multitier approach increases the performance, scalability, and reliability of an application. EJBs also increase flexibility, since they can be quickly modified to meet changing business rules. The distributed approach makes them location-independent, so a system administrator can move them around to reconfigure system load.

On the horizon

As Java picks up momentum and a greater mindshare, it will become more and more accepted as "enterprise-ready." Looming on the horizon are Enterprise JavaBeans. EJBs are transaction-ready, scalable JavaBeans, paving the way for doing heavy-duty enterprise applications in the NCF. JDK 1.2 promises to add performance as well as a slew of additional APIs. The Java Servlet Development Kit (JSDK) will be a standard extension of 1.2 as well as JIDL, the Java interface definition language, for supporting CORBA objects, JMAPI, the Java Management API, and JNDI, the Java Naming and Directory Interface.

Benefits of Java for the server-side programming model. Reasons for choosing Java as the basis of the NCF programming model are very concrete. There is definitely goodness in the fact that Java has a lot of mindshare in the Internet space, but in addition to that, many technical reasons make Java a compelling choice for NCF applications. These reasons are given in the following subsections.

Portability. Although the Java promise of "write once, run anywhere" is not yet completely fulfilled, Java is by far the most portable language yet. There are still differences in the Java Virtual Machines on different platforms and by different vendors, but these differences are slowly going away. Sun's "100 percent pure Java" initiative and set of comprehensive test suites are making the bridge between platforms and vendors smaller and smaller. Although some have suggested that the Java promise is "write once, test everywhere," that is still much more productive than the porting nightmares associated with C and C++ applications.

Functionality. Java is not only a language, but more and more a programming environment. Java is building in such powerful function as JDBC, JavaBeans, and JTS, making it a very functional and powerful development platform.

Performance. Because servlets are persistent, reducing startup and destroying overhead, and because of the fact that they run in the same process as the Web server, they typically run several times faster than CGI programs.

Security and reliability. The Java Virtual Machine restricts servlets from accessing server resources, and through the use of a Security Manager, an administrator can impose restrictions on running servlets, making the Web server more secure and reliable. Current Web server APIs such as Internet Server API (ISAPI) and Netscape Server API (NSAPI) could crash and corrupt the hosting Web server. The built-in security mechanism of Java is now considerably revamped with the introduction of the JDK 1.2 security model.

Consistency. One of the primary goals of the NCF was to follow consistent industry standards. The "Web revolution" was spurred by interoperability, based on a simple standard (HTML and HTTP). Java has shown great promise as a consistent standard; the 100 percent Java program ensures interoperability on all platforms; and comprehensive test suites exist to ensure that vendor implementations of the virtual machine adhere to the JavaSoft specification.

Concluding remarks

The use of Java on the server is gaining momentum. It is proving to be very convenient, portable, and productive. Server-side Java APIs such as servlets and Enterprise JavaBeans are increasingly prevalent and becoming common building blocks of serious business applications. IBM is investing in this vision by basing its WebSphere Application Server on this technology, and partnering with JavaSoft to build the best servlets engine. This paper touched on the most critical components of this architecture, the major pieces of this server-side programming model—servlets, JSP files, session state, JavaBeans, and Enterprise JavaBeans. The paper has provided an overview of this subject area.

Acknowledgment

I would like to thank Mark Fisher, Tricia York, and the rest of the NCF architecture team for their help in editing the draft of this paper.

Cited references

1. I. F. Brackenbury, D. F. Ferguson, K. D. Gottschalk, and R. A. Storey, "IBM's Enterprise Server for Java," *IBM Systems Journal* **37**, No. 3, 323–335 (1998, this issue).
2. *JavaSoft Java Server, JavaDocs* (http://jserv.javasoft.com/products/java-server/documentation/index.html), Sun Microsystems, Inc.

Accepted for publication April 30, 1998.

Elias Bayeh *IBM Software Solutions Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: ebayeh@us.ibm.com).*

Mr. Bayeh joined IBM in 1995, working as the technical lead on the IBM (now Lotus Domino Go) Web Server. He is now the technical and design lead on the WebSphere Application Server and the NCF programming model. Currently a senior engineer, Mr. Bayeh received a B.S. in computer science from Truman University in 1984 and completed all requirements for a B.S. in business administration in 1990, also from Truman University. He has filed numerous patents pertaining to the use of Java on the server, and led the authoring of the NCF Architecture Workbook.

Reprint Order No. G321-5680.