J2EE

A Distributed Component Architecture Model

By
Mike Brevoort, Sr. Associate
Mike Steinley, Sr. Associate
Heather Spinnenweber, Sr. Associate
Topics covered in this paper include:

- Distributed Component Models
- The J2EE Component Model
- J2EE Architectural Overview
- J2EE: A Platform for Enterprise Web Services

Overview

Approximately 70 percent of Global 2000 enterprises use Java technology, and by the year 2006 the number of professional Java developers worldwide is projected to grow to 2.5 Million (Gartner). Adopted by the open-source community and employed by industry goliaths, Java and its enterprise component framework, Java 2 Enterprise Edition (J2EE), has emerged as a leading component architecture for applications requiring agility, scalability and interoperability. In this whitepaper, we provide an overview of the J2EE architecture and its emergence as an enabling component of Enterprise Web Services, a key element of an intelligent Enterprise Architecture.

Introduction

The increased importance of technology in business strategy has forced organizations to gravitate toward more agile, flexible, and scalable methods for designing and implementing business applications. The simultaneous emergence of the Web and the increased attention to object and component development models has resulted in rapid adoption of distributed component architectures and their associated tools and platforms.

Distributed component architectures provide the foundation and standards for invoking objects within one application from another application independent of application location. The applications could reside on the same server, different servers, or even in different organizations.

I. Distributed Component Modules

Distributed component architectures such as J2EE are promising in two ways. They facilitate application integration because organizations can use component models to provide common interfaces for disparate applications that need to be integrated. Secondly, they enable rapid and flexible application development, shifting effort from writing code to assembling and integrating pre-built modules, thus reducing the cost and time to develop new applications.

A component is defined as a collection of related functions that encapsulate data. These functions are then put into a package that can be called by a program to perform a task or event. The physical location of the program calling the component is irrelevant. For example, the program could be on the other side of the globe at a business partner's office. The component's interface describes the functions that can be performed as well as the format for inputs and outputs via an interface definition language (IDL) that enables components to communicate with each other. Regardless of the programming language, the common IDL ensures that different components can communicate with each other. Components must reside in an environment that can locate the components and pass messages among them. The combination of the IDL and this component environment, with its associated services and utilities, is referred to as a component model.
The best known component models are Microsoft’s Distributed Component Object Model (DCOM) and J2EE’s Enterprise Java-Beans (EJB). Figure 1 shows an example of the basic components of a component system: Software components, a standard interface, and common services.

![Figure 1: The J2EE Component Model](image)

The Microsoft DCOM approach allows developers to use any programming language to build applications, but the applications themselves will run only on the Windows operating system. By contrast, J2EE’s approach requires that applications be built using the Java programming language, though they can run on any operating system.

II. The J2EE Component Model

The J2EE platform provides a unified approach for enterprise-wide, distributed applications through a component-based application model. J2EE defines a standard framework for developing multi-tier enterprise applications that are highly available, secure, and scalable. This framework aims to achieve these benefits by defining a standard architecture outlined by complete specifications and compliance tests to ensure portability of applications across the wide range of existing enterprise systems capable of supporting J2EE. These specifications are created and maintained by the Java Community Process (JCP). The JCP was founded by Sun in 1995 and has evolved into a formalized process overseen by representatives from many organizations across the Java community. It is an open, inclusive organization of active members. The JCP holds the responsibility for the development of Java technology, guiding the development and approval of Java technical specifications.

A wide variety of technology vendors are the key suppliers of J2EE compliant platforms, tools, and products. These vendors span the technology universe and include operating system vendors, database system vendors, application server vendors, and Web server vendors. A vendor must make J2EE APIs available to the application components through containers and must provide the mapping of the application components to the network protocols as laid out by the J2EE specification. A J2EE product is free to implement interfaces that are not described by the specification in an implementation-specific way. This
allows each vendor to distinguish themselves by creating integration and add-ons to their product suite. A J2EE vendor also must provide application deployment and management tools, though these tools are not prescribed by the J2EE specification.

III. J2EE Architectural Overview

The J2EE platform is essentially a distributed Java application server environment that provides a set of Java extension APIs to build applications and a runtime infrastructure for hosting applications. The most significant aspect of J2EE is its abstraction of the run-time infrastructure. The specification does not mandate how a J2EE runtime should be built; instead, it specifies roles and interfaces for applications and allows the runtime to abstract most of the infrastructure services that enterprise developers have traditionally attempted to build on their own. As a result, application developers can focus on application logic while leveraging the runtime for all infrastructure-related services.

J2EE unifies access to enterprise services such as transaction processing, database access, and messaging, through its enterprise service APIs. Containers provide access to each implementation of a specific device via the APIs. Again, the interfaces defined by the J2EE platform specification and the implementation are left to the vendor. J2EE specification defines several major elements of the architecture:

- Containers and Application Servers – Runtime environments that provide services to J2EE components.
- J2EE Components – Application constructs whose lifecycles are managed by containers.
- Enterprise Service APIs and J2EE Connector Architecture – Integration facilitators to the Enterprise Information System (EIS) tier and other middleware.
J2SE – Java 2 Standard Edition

J2SE is, in the simplest terms, the Java development environment. It includes the Java Virtual Machine (JVM), the core and integration APIs, user interface toolkit, and development toolkit.

The JVM is the runtime engine that interprets Java Byte Code into platform specific instructions, facilitating platform independence and modularity. Core and integration APIs include data structures, network communications, database and localization. Connectivity, I/O, and internationalization.

Containers and Application Servers

A J2EE container is a runtime that manages application components and provides access to the J2EE APIs. The runtime manages component lifecycle, security, deployment, and runtime services. Introducing a container layer around components allows the components to transparently access J2EE services such as transaction management, security checks, resource pooling, and state management without regard to a vendor's specific service implementation.

J2EE specifies four containers – a Web container for hosting Java Servlets and JSP pages, an EJB container for hosting Enterprise JavaBean components, an applet container to run applets, and an application client container for running standard Java application clients. We limit our discussion here to Web and EJB containers only. Application servers provide the infrastructure for developers to manage information, represent business logic, and interact with users while isolating the users from the fact that these systems
may be composed of a variety of internal applications. These Application Servers generally provide core object and session management functionality through the use of Object Containers.

A J2EE application server vendor typically implements the J2EE server-side functionality using an existing transaction processing infrastructure in combination with J2SE technology. An application server is an environment for containers, providing runtime services such as clustering, fail-over, load balancing, and inter-container communication.

**J2EE Architectural Elements**

J2EE defines three building blocks that are used to create component-based enterprise applications. These elements are constructs that hold presentation and business logic on the middle tier facilitating good system design. The three primary architectural elements of J2EE are Servlets, Java Server Pages (JSP), and Enterprise JavaBeans. The logical relationships of the architectural elements of the J2EE platform are shown in the figure 5.

A Servlet is a Java object that extends the functionality of an HTTP server. As the entry point into an n-tiered application, it generates dynamic content, responding to clients’ HTTP requests with corresponding responses. Designed as a better alternative to CGI, NSAPI, and ISAPI, the Servlet is more efficient, provides inherent session management, and is platform/server independent. A Servlet executes within a Web container.

Built on Servlet technology, JSP combines markup (HTML or XML) with “scriptlets” of inline Java code to produce dynamic content. When a user requests a JSP page, the Web container compiles the JSP page into a Servlet. The Web container then invokes the Servlet and returns the resulting content to the client; therefore JSPs are a powerful and dynamic page assembly mechanism. Compared to Servlets, which are composed of pure Java code, JSPs are merely ASCII text documents until the Web container compiles them into Servlets. JSPs are extensible via custom tag libraries, enriching functionality while maintaining simplicity. JSP tags are Java components that can be used from within a JSP file. By representing these components as tags, JSPs enable content developers to implement the presentation layer separate from
the business logic. Simple JSP tags can be developed improving reuse and transparently adding complex functionality to the presentation layer.

One of the richest components of the J2EE framework is the Enterprise JavaBean (EJB). The EJB component model simplifies the development of middleware applications by providing automatic support for services such as persistence, transactions, security, database connectivity, and more. EJB is a distributed component model for developing portable, distributable, transactional, and scalable components existing over heterogeneous J2EE servers and operational environments. More simply, EJBs are a reusable bundle of business logic. Just as JSPs allow the separation of application and presentation logic, EJBs allow separation of application logic from system-level issues, allowing the developer to concentrate on the business domain issues rather than system programming. An EJB container manages the lifecycle of an EJB and shields the bean developer from issues including transactions, security, scalability, concurrency, communication, resource management, persistence, error handling, operating environment, and independence.

Enterprise JavaBeans implement security through declarative access control. Access is granted based on configured security roles defined within the EJB container. The application server will provide deployment tools that map these abstract roles to users or groups of users that actually exist in the enterprise’s security environment. For example, the ‘fund_administrator’ role might be mapped to everyone who is a member of the Windows security group ‘Fund Manager.’ In this example, only members of the ‘Fund Manager’ group may call an EJB business method with a method permission granted to the ‘Fund Manager’ group. At a high level, there are three varieties of Enterprise JavaBeans – entity beans, session beans, and message-driven beans.
Entity, Session, and Message-Driven EJBs

- Entity Beans are persistent objects that are object-oriented representations of data in a database. Like a database, multiple clients can access it simultaneously. An entity bean might represent an account, product or person. The lifespan of an entity bean is exactly as long as that of the data it represents and the bean is logically the single point of access for that data. EJB containers can manage the process of saving and restoring entity bean state; this is known as container-managed persistence (CMP). Alternatively, developers can control the bean’s persistence themselves; this is known as bean-managed persistence (BMP). CMP promises large boosts in productivity and performance eliminating the need for developers to write and maintain SQL.

- Session beans encapsulate business logic or workflow and may be stateful or stateless. Stateful session beans can keep data between client accesses, a transient object such as a shopping cart. Stateless session beans may be used to implement a specific service such as a quick database update and have request-scope, basically a distributed static method call. Session beans execute on behalf of a single client.

- The Enterprise JavaBeans 2.0 specification added a new type of EJB, called a message-driven bean, which can act as a listener for Java Message Service API (JMS), processing messages asynchronously. A message-driven bean is similar to a stateless session bean, though it is exclusively as a “boundary object,” in that it is an entry point into the business logic or middle tier.

The EJB specification is designed for the task of representing distributable business objects without making the business-logic developer provide system level services. Though EJBs may be overkill for the average dynamic Web-based application, any enterprise level architecture will benefit from writing business logic as EJB components, opening up new possibilities in developer productivity, application deployment, performance, reliability, and reusability.

Enterprise Service APIs and the J2EE Connector Architecture

The J2EE architecture unifies access to enterprise services through its enterprise service APIs. Instead of having to access these services through proprietary or non-standard interfaces, J2EE application programs can access these APIs via the container. Connectors provide standard access to external enterprise resources including databases and other EIS resources. Below is a summary of these resources and their roles with a J2EE distributed application

- JDBC – The Java Database Connectivity API provides connectivity to relational database systems allowing transactional querying, retrieval, and manipulation of data from a JDBC-compliant database. The JDBC API comes with the J2SE but J2EE adds advanced features such as connection pooling and distributed transactions.

- JNDI – Java Naming and Directory Interface API provides standard access to a directory services resource such as LDAP or Novell Directory Services. Secondly, J2EE uses JNDI to look up registered services or distributed components such as data sources or EJBs, respectively. JMS – The Java Message Service provides functionality to send and receive messages asynchronously through the use of message-oriented middleware. This is critical in an enterprise environment where the various distributed components may not always be in constant contact with each other.

- JavaMail – JavaMail supports the most widely used Internet mail protocols such as IMAP, POP, and SMTP and is also used to send and receive asynchronous messages, but is more oriented towards the user rather than parts of an application. Compared to JMS, it is slower and less reliable.
RMI – The Remote Method Invocation allows methods to be called on remote, distributed Java objects as if they were local. It is the primary mechanism used by the Enterprise JavaBeans framework for inter-process method invocation.

RMI-IIOP – An extension of RMI but over Inter-ORB Protocol that allows you to define a remote interface for OMG CORBA objects and invoke methods through JavaIDL.

The J2EE Connector Architecture (JCA) is the standard architecture for connecting the J2EE platform to heterogeneous Enterprise Information Systems (EIS). By defining a set of scalable, secure, and transactional mechanisms, the J2EE Connector architecture enables the integration of EISs with application servers and enterprise applications.

JCA defines the following contracts between the application server and the application to be integrated:

- A security contract, so that the J2EE application server can securely access the external application to be integrated
- A performance contract, able to pool connections with the application to be integrated, to optimize performances
- A transactional contract that will enable the external application to participate in a distributed transaction.

Though JCA is still in its infancy, tremendous progress has been made toward standardizing interface connectivity to enterprise resources. Before JCA, EIS integration was done as a specific point-to-point implementation or facilitated by a third party, proprietary integration broker. These implementations are
mostly based on message-oriented or ORB middleware and offer no standard support for distributed transactions or connectivity. JCA promises standard interface connectivity and the ability for each resource to participate in distributed transaction.

J2EE: A platform for Enterprise Web Services

The simplicity of distributed component architectures can be undermined when organizations find themselves supporting two or more approaches to distributed component models such as the Microsoft’s DCOM/.Net, J2EE, or OMG’s CORBA. The integration of multiple component models necessary to provide interoperability becomes a major challenge. For this reason, excitement has been growing around Web Services, a set of technologies and approaches based on Extensible Markup Language (XML) as a component model for Internet communications supporting component architecture independence. Both Microsoft and Sun continue to evolve their component-based architectures to leverage XML, with Microsoft introducing .Net and Sun rapidly extending the J2EE specification. In fact, the most visible change to the J2EE 1.4 platform is strong integration of Web Services. J2EE components can now interact with Web services and implement them in a standard, portable fashion.

XML and other Web Services protocols such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI), are standards endorsed by the World Wide Web Consortium (W3C). These standards are being rapidly implemented by almost every significant vendor, resulting in important progress toward the promise of inter-enterprise application integration.

Since its birth, J2EE has leveraged XML in its runtime environment for configuration and deployment. Now with the emergence of Web Services, Java has expanded its XML integration offering, providing a rich suite of functionality. Similar goals including platform independence, portability, and vendor neutrality sensibly promote a marriage of the two technologies and are consistent with the overall J2EE platform goals. Web Services will play a pivotal role in enterprise application architecture and integration, enabling implementation independent interoperability. Through its strong XML capabilities, J2EE is poised to immediately enable Web Service development and integration and is aligned with Java’s founding principle of platform independence. The J2EE Web Service will be a key integration component in next generation applications.

The J2EE framework, coupled with architectural design patterns, is positional to provide the foundation and building blocks for tomorrow’s enterprise applications. The portability, interoperability, reliability, and stability of Java provides a flexible foundation for the automation of many repeatable processes, and J2EE’s strength is in its ability to work across heterogeneous systems both within the enterprise and the extended enterprise.

For today’s organizations, it is increasingly difficult, to find only one component model (.NET or J2EE). The reality is that both frameworks will co-exist within an enterprise architecture. Organizations must understand how and when to use the appropriate model and not invest effort in attempting to realize the unrealistic vision of a homogenous component architecture. A clear understanding of J2EE and its place in a Web Services model can help an enterprise to get the most out of its Java-based technology assets.
References
J2EE 1.3 Specification
EbizQ, “Which Technology for Tomorrow’s EAI?”, by TechMetrix Research 3/18/02
PWC Technology Forecast 2001-2003
PWC Technology Forecast 2002-2004
Figure 2 (Source: PWC Technology Forecast 2001 – 2003)
Figure 3 (Source: Sun, 2002)
Figure 4 (Source: PWC Technology Forecast 2001-2003
Figure 6 (Source: ebizq.net)

About the Strategic Technology & Advancement Research (STAR) Team
The Strategic Technology & Advancement Research Team serves as the proving ground for our ideas.

STAR brings together the best and brightest of LiquidHub to encourage new thinking and develop through leadership around technology. Breakthrough ideas emerge and are exercised by the Team, to strengthen client engagements and educate industry leaders.