Patricia Seybold Group

Strategic Technologies, Best Practices, Business Solutions

Selecting Enterprise JavaBeans Technology

By Anne Thomas

July 1998

Prepared for WebLogic, Inc.

85 Devonshire Street, 5th Floor, Boston, MA 02109 • Phone 617.742.5200 • Fax 617.742.1028 •
www.psgroup.com

Table of Contents

Illustrations

Selecting Enterprise JavaBeans Technology

By Anne Thomas, Patricia Seybold Group
July 1998
Prepared for WebLogic, Inc.

Executive Summary

Enterprise JavaBeans

Enterprise JavaBeans (EJB) provides services to network-enable applications so that they may be easily deployed on intranets, extranets, and the Internet. EJB is a standard server component model for Java application servers. The component model allows server-side Java application components to be deployed in any application server that supports the EJB specification. EJB supports a new approach to application development based on rapid component assembly. EJB server components are reusable, prepackaged pieces of application functionality that can be combined with other components to create customized application systems.

Simplifying Development

EJB also provides an integrated application framework that dramatically simplifies the process of developing enterprise-class application systems. The EJB server automatically manages a number of tricky middleware services on behalf of the application components. EJB component builders can concentrate on writing business logic rather than complex middleware. As a result, applications are developed more quickly and the code is of better quality.

Universal Support for EJB

The EJB server component model is completely vendor-independent. Any application server vendor can add support for EJB components, and in fact, most vendors are doing just that. The EJB specification was published in March 1998. Within minutes, fifteen application server vendors announced plans to implement support for EJB. EJB-compliant application servers are available now, and the application assembly process works. By the end of 1998 there should be about a dozen products to choose from.

WebLogic Tengah

WebLogic was the first Java application server vendor to release full support for EJB. Tengah is a highly scalable, general-purpose application server that supports both Web-based and desktop applications. Tengah was specifically designed to support the application component assembly approach, and the system provides a very clean implementation of the EJB specification. For those looking to reduce costs, improve quality, and build applications quickly (and who isn't?), EJB is the way to go. Tengah provides an excellent EJB host.

Enterprise JavaBeans Technology

Java Application Servers

Enterprise JavaBeans (EJB) technology is an integral component of Sun's Java Platform for the Enterprise (JPE). EJB defines a standard set of programming interfaces for a Java application server. The standard interfaces are based on server component technology, which supports rapid development and reusability. The EJB component model dramatically simplifies enterprise-class application development by automating complex middleware services. The EJB model also defines a standard programming API to allow application components to be ported and used in any application server without code modification.

Combining the Best of the Mainframe, the Web, and Client/Server

JPE and EJB together form a powerful enterprise operating environment that combines the best features of centralized mainframe-based computing with the Web and client/server. Illustration 1 provides an overview of a typical JPE environment. An EJB server can support traditional desktop LAN clients, Web clients, and Internet appliances and devices. Application logic is deployed as server components within an EJB application server. Components can access both new and legacy environments, including applications, databases, files, datafeeds, and unstructured data.

Implementing EJB

Sun Microsystems published the EJB specification in March 1998, and within minutes, fifteen Java application server vendors had announced plans to implement support for EJB in their products. The EJB specification defines a very comprehensive model, but it does not define implementation details. Therefore vendors have the opportunity to differentiate their products through special features and services.

Key Requirements

This paper defines the key requirements of an EJB application server and provides sufficient evaluation criteria to enable a discriminating buyer to select an appropriate Java application server that fits the needs of a particular organization. Table 1 outlines the key characteristics that should be available in an EJB implementation. These characteristics are defined in detail in the remainder of the report.

Typical JPE/EJB Environment

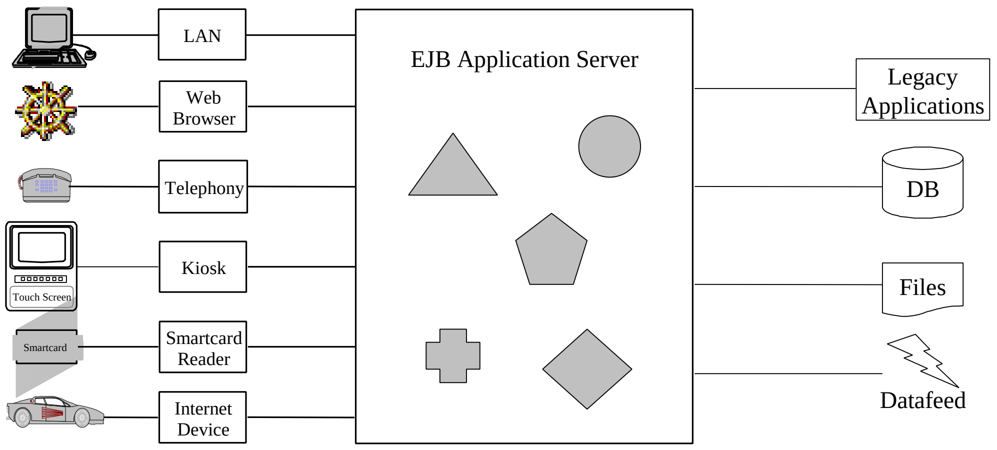

Illustration 1. A typical JPE/EJB environment can support a variety of client environments, including desktop clients connected through a LAN, Web browser clients, telephony clients, kiosks, smartcards, and other Internet devices. Business logic is implemented as reusable application components deployed in the EJB application server. These components have complete access to all types of data and applications, including relational and non-relational databases, flat files, live datafeeds, unstructured data, and legacy applications.

Requirement	**Reasoning**
Comprehensive EJB Support	EJB dramatically simplifies development and supports portability and reusability. An EJB server should provide full support for all EBJ features and services to ensure the most benefit.
Comprehensive JPE Support	JPE defines a standard set of interfaces to a variety of middleware services. These interfaces can be used to network-enable applications, uniting web, database, and distributed components. JPE supports portability and reusability. EJB does not require full support for all of the JPE APIs, but the portability and reusability benefit is compromised if an application server uses proprietary APIs.
Non-Proprietary	An EJB server and the EJB applications that run within it should be completely independent from any operating system, Java platform, database, development tool, client connection, integration service, or middleware infrastructure. Proprietary technologies limit flexibility, portability, and reusability and ruin any chance of using off-the-shelf application components.
Robust Environment	Enterprise applications require a robust execution environment that supports high performance and scalability, comprehensive security, and bulletproof reliability.
Comprehensive Management	Distributed application systems consist of many small components that can be scattered across many different systems. The management requirements for this type of environment are more complex than for any others. A conceptually centralized management system that coordinates the entire distributed environment is required.

Table 1. The key characteristics that should be supported by an EJB server.

Introduction to Enterprise Java

Java Platform for the Enterprise

Supporting Enterprise-Class Applications

Sun Microsystem's Java Platform for the Enterprise (JPE) provides a foundation for the development and deployment of Java-based enterprise-class application systems. JPE elevates Java to a serious application development language capable of supporting mission-critical distributed enterprise application systems. Enterprise JavaBeans (EJB) is an essential piece of the complete JPE environment. EJB defines a model to support the development and deployment of server-side Java application components.

Enterprise-class Computing

The term enterprise implies an extremely robust application environment. Enterprise-class computing requires the highest caliber scalability, availability, reliability, security, and integrity. In order to achieve this level of performance and quality of service, enterprise applications require access to a core set of distributed infrastructure services, including management, naming, security, transactions, and persistent data stores.

JPE APIs

The Java Platform for the Enterprise defines nine Java APIs that enable Java applications to access the above mentioned core enterprise-class infrastructure services through a set of standard programming interfaces. Table 2 briefly describes the APIs. The JPE APIs also enable development, deployment, and management tools to more easily integrate with infrastructure services and application servers. Unlike most standards efforts, JPE does not require a new set of infrastructure services. The JPE APIs were designed to make use of the infrastructure services that are already in place. The JPE APIs simply provide a new programming interface to the existing services.

Portability and Productivity

The JPE APIs, and the EJB API in particular, are designed to deliver two critical benefits: portability and productivity. The JPE APIs allow enterprise applications to be ported from one set of infrastructure services to another without modification. The EJB API extends the portability model to also include application servers.

Simplifying Middleware

The EJB API provides numerous automatic services that significantly simplify the enterprise application development process. Developing well-behaved enterprise-class application systems that perform and scale is quite hard. Developers often spend more time writing middleware code than business logic. The EJB API completely automates many complex middleware services, such as naming, distribution, lifecycle, transactions, security, state management, and persistence. With EJB, developers can concentrate on the business rather than the middleware.

API	Description
EJB	Enterprise JavaBeans is a server component model that provides portability across application servers and implements automatic services on behalf of the application components.
JNDI	Java Naming and Directory Interface provides access to naming and directory services, such as DNS, NDS, LDAP, and CORBA naming.
RMI	Remote Method Invocation creates remote interfaces for Java-to-Java communications.
Java IDL	Java Interface Definition Language creates remote interfaces to support Java-to-CORBA communications. Java IDL includes an IDL-to-Java compiler and a lightweight ORB that supports IIOP.
Servlets and JSP	Java Servlets and Java Server Pages support dynamic HTML generation and session management for browser clients.
JMS	Java Messaging Service supports asynchronous communications through various messaging systems, such as reliable queuing and publish-and-subscribe services.
JTA	Java Transaction API provides a transaction demarcation API.
JTS	Java Transaction Service defines a distributed transaction management service based on CORBA Object Transaction Service.
JDBC	JDBC Database Access provides uniform access to relational databases, such as DB2, Informix, Oracle, SQL Server, and Sybase.

Table 2. The Java Platform for the Enterprise APIs.

Portability Across Infrastructures

One of the primary tenets of Java is to support application portability. Java's universal theme is "write once, run anywhere." The Java virtual machine (JVM) provides a common execution environment that allows a Java application to run on any operating system. But the distributed infrastructure services are often separate from the underlying operating systems. Organizations can use any number of different products and configurations from multiple vendors to implement the infrastructure services. The JPE APIs extend the JVM, enabling applications to be completely portable across multiple infrastructures.

For example, the Java Naming and Directory Interface (JNDI) API allows a Java application to use one common interface to access any naming or directory service. A Java application can use an LDAP directory on one system and a CORBA naming service on another system, without changing any code.

Application Servers

One of the most critical aspects of enterprise computing is scalability. The Internet has added a new dimension to the concept of high-volume application systems. Internet applications may need to support hundreds of thousands, if not millions, of concurrent users, therefore enterprise Java applications will often be required to scale to unprecedented proportions. The most effective way to achieve scalability is by distributing the processing load across multiple processors while efficiently managing and recycling scarce system resources. Scarce system resources include operating system processes and threads, database connections, and network sessions. All of these resources are expensive to create and, once created, consume large amounts of memory. To achieve optimal performance, these resources should be created once and then saved and recycled for subsequent requests. An application server provides a runtime environment that recycles these scarce and expensive system resources.

Emerging JAS Market

A growing number of vendors are now offering application servers specifically designed to support Java applications. Application servers can come in a variety of different forms, such as Web servers, TP Monitors, or database servers. Each Java application server (JAS) offers a slightly different set of capabilities, so selecting a vendor requires some investigation and analysis. As a general rule, a JAS provides an optimized execution environment that supports automatic management of scarce resources. But each product offers different advantages and disadvantages. For example:

- Some systems offer simplicity while others offer extremely sophisticated tuning features.
- Some systems provide integrated development tools.
- Some systems supply special data access or persistence services.
- Some systems are available on limited platforms while others provide broad platform support.
- Some systems focus purely on Internet applications while others support multiple client environments.
- Some application servers support only Java while others support multiple languages.
- Some systems support bulletproof fault tolerance and recoverability.
- Some systems will scale more than others.
- Some systems use proprietary APIs while others support portability through JPE and EJB.

Proprietary APIs

Traditionally, application servers have always been proprietary. Each application server would supply its own APIs enabling the application server to control and manage the execution of the server applications. An application developed for one application server could not be ported to another application server without extensive rewrites. For example, an application developed for Tuxedo can’t run in Microsoft Transaction Server.

Standard APIs

JPE aims to support complete portability—across platforms, across infrastructure services, and even across application servers. By using the JPE APIs, an application can operate in any environment. The JPE APIs define a standard vendor-independent interface between a Java application and any middleware service. The Enterprise JavaBeans specification defines a standard vendor-independent interface between a Java server component and any Java application server. Any EJB-compliant application can execute in any vendor's EJB-compliant Java application server.

EJB Value Proposition

EJB Benefits

Portability provides vendor independence, and while many users appreciate this benefit, it probably isn't enough to convince the world to adopt Enterprise JavaBeans. But EJB provides quite a few other benefits related to reusability and increased productivity.

Business Object Reusability

Since the first inception of object-oriented technology in the late 1960s, object proponents have promised code reusability. According to the vision, business objects providing discrete application functions will be available from a variety of vendors. Users will be able to buy these off-the-shelf objects and rapidly assemble customized business applications.

Components

To a certain extent, this vision is already a reality. The vision is made possible by components. Components are objects that are designed specifically to support reusability. Unlike class libraries and other objects, components can be customized without modification of the source code. Components support codeless customization through a set of properties that are external to the code within the class. Therefore a single object source can be used in a variety of ways in different applications. This approach reduces the effort required to manage, test, debug, and maintain objects.

Development Components

Quite a few vendors provide reusable development components, such as ActiveX controls and JavaBeans. Development components are reusable widgets and tools that can be used in numerous application development tools. Development components range in functionality from simple buttons to complex spreadsheets or financial calculators.

Discrete Business Functions

But development components only provide a portion of a complete business function. According to the vision, a business object supplies a discrete business function, such as order processing or account management.

Frameworks

Thus far, the plug-and-play business object marketplace has been slow to develop. To date, the most popular approach to delivering server-side business object reusability is to use frameworks. A framework is a skeletal implementation of an application that provides a core set of functions and capabilities. The application skeleton is based on a predefined object model, and it implements a set of interrelated abstract classes to represent the object model. Developers flesh out the skeleton by subclassing the abstract classes and writing custom business methods. A framework saves a lot of time compared with building an application from scratch, but the customization process can still take quite a bit of time.

Reusable Server Components

EJB applies the reusability characteristics of components to server-side business objects. EJB enables software developers to build application functionality into small-grained business objects that can be used and reused in a variety of configurations.

Server Component Advantages

EJB server components are reusable, prepackaged pieces of application functionality that can be deployed in an EJB-compliant application server. EJB server components offer a tremendous number of productivity and flexibility advantages over frameworks, including:

- **Store-Bought Business Objects.** Server-side components offer complete business object functionality. A framework is generally only 40–60 percent complete, while a server component is generally ready for deployment. Assuming that the component comes from a reputable vendor, the code will be fully tested, debugged, and ready to go.
- **Customization.** As with development components, developers can customize the behavior of a server component through properties without modifying the source code. Although there are limitations to the customization capabilities available through a property table, the environment can be very flexible. For example, properties could be used to define customizable business rules.
- **Plug-and-Play Application Assembly.** Server components from different vendors can be combined to create custom application systems. Components could be sold as individual business objects, as a group of related business objects, or as a complete integrated application.
- **Deployment Options.** Because EJB supports complete portability across application servers, EJB users have limitless deployment options. As application requirements grow and change, components can be redeployed onto larger and more powerful systems.

Productivity Advantages

Obviously, using store-bought business objects is more productive than building an entire application from scratch. But the productivity benefits don’t end there. Perhaps the most valuable benefit of Enterprise JavaBeans is that the environment makes it much easier to build sharable, scalable, distributed application systems. Enterprise-class application systems are much harder to develop than simple client/server applications. In most cases, enterprise applications must be highly scalable, reliable, secure, and transactional.

Implicit Services

An EJB application server supplies a set of implicit services that automatically manages the complex issues of distributed object computing. Lifecycle, multithreading, load balancing, session state, fault recovery, concurrency, synchronization, transactions, security, and persistence are all managed by the EJB application server on behalf of the application component. Enterprise JavaBeans greatly reduces the amount of code that needs to be developed in each

Java server component. EJB developers write simple Java classes as if they were to be used by a single user. EJB then ensures that the components behave properly in the shared, multi-user environment.

Attribute-Based Semantics

EJB uses an innovative technique to define the semantics of a component's behavior. One of the more challenging issues that stymies business object reusability is allowing a business object to behave differently in different situations without modifying the business object code. Component systems support codeless behavior modification using attributes. The semantics of an application can be defined in a set of attributes that are separate from the application code. By separating the semantics from the application code, a single component can behave differently in different applications. A user of the component can change the behavior of the component by changing the value of the attributes when the components are assembled into an application system and deployed in an application server. EJB provides a standard mechanism to use attributes to dynamically define lifecycle, transaction, security, and persistence behavior in EJB applications.

Illustration 2 further explains this concept. The *Account* object represents a consumer bank account. The object supports fairly straightforward banking operations such as *GetAccountBalance*, *Deposit*, and *Withdraw*. In the first scenario, a client application interacts directly with the Account object. In this case, the Account object starts and completes a transaction on each method call. In the second scenario, a *Teller* object provides an interface between the client and the Account object to implement added control and add support for a *Transfer* operation. In this case, the Teller object starts the transaction and the two Account objects enroll in the transaction started by the Teller object. Because the transaction semantics of the Account object are defined as attributes, the same Account object can be used in both scenarios without modification of any application code.

Integrating with Legacy Systems

Component technology is also very useful as a means to integrate new application systems with the legacy environment. Components can be used to build reusable interfaces to existing applications, such as CICS mainframe transactions or enterprise resource planning (ERP) systems. Through EJB, these backend systems could be made accessible across the Internet to support better client services or partner relations. This approach enables incremental evolution of application systems rather than whole-scale change and replacement.

EJB Is Fostering the Component Marketplace

For the moment, the market for server-side components is still very young. Until recently, it wasn't feasible for vendors to build server components. Each application server required the use of its own proprietary APIs, so vendors were required to develop applications for one specific deployment environment. For the most part, business application vendors built applications for only the most popular application server environments, such as CICS or Tuxedo. But the EJB portability model gives vendors and users much better options. EJB components can be deployed in any EJB-compliant application server, and most application server vendors are implementing support for EJB.

Changing Transaction Semantics

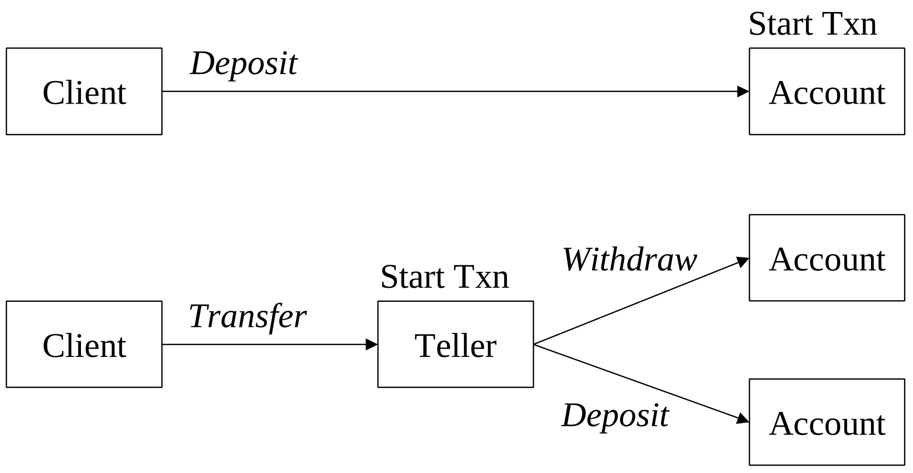

Illustration 2. Using attributed-based transaction semantics, the same Account object can exhibit different transaction behavior when used in different applications. In the first example, the Account object starts a transaction (Txn). In the second example, the Account objects join a transaction started by the Teller object. No code modification is required within the object to affect the change in transaction semantics.

Rapid Adoption

The EJB specification was finalized in March 1998, and within days most Java application server vendors and industry leaders (including IBM, HP, Sun, Oracle, Sybase, and Novell) endorsed EJB. Within four months, products began shipping. By the end of 1998 there will be more than a dozen EJB application servers on the market. With all the system and application server vendors committed to EJB, the business application vendors are beginning to implement applications and components using EJB.

Evaluating an EJB Server

Java Application Servers

A Java application server supports the execution of server-side Java applications or components. Java application servers can come in a variety of forms, such as general-purpose application servers, object servers, component servers, web servers, database servers, or transaction servers. Not surprisingly, Java application servers provide different services and capabilities.

JPE Support

Although the Java Platform for the Enterprise is still quite new, we consider JPE API support to be a critical feature in Java application servers. We recommend that organizations avoid using a Java application server that uses proprietary APIs or requires a proprietary Java development tool. Not only will a proprietary environment cause vendor lock-in, but it will also severely limit the number of third-party applications and components that can be used with the environment.

EJB Support

In particular we recommend using a Java application server that supports the EJB component model. EJB automates a number of tricky application services, dramatically simplifying enterprise Java development and increasing developer productivity. We predict that a large market for plug-and-play server-side components will grow around EJB, enabling users to develop and deploy customized application systems more quickly than ever before.

Integrated Application Server

We also recommend using a complete, integrated Java application server that combines and maximizes the capabilities of EJB and the other JPE APIs to provide the best possible performance and reliability. An application server should fully integrate load balancing, failover, security, transactions, persistence, and management services with the EJB and JPE environment.

Differentiating Enterprise JavaBeans Servers

EJB defines a standard set of interfaces that enable an application component to run in any application server. By definition, an application server manages resources on behalf of its application. But the exact nature by which the application server manages these resources is not defined within the scope of the Enterprise JavaBeans specification. Individual vendors can differentiate their products based on the simplicity or sophistication of the services. The following sections provide evaluation criteria that can be used to measure the different vendor offerings.

Comprehensive Support for EJB

What Is EJB Support?

Most Java application server vendors have announced plans to support EJB. A small number of vendors are releasing EJB support during Q3 1998. Quite a few more will implement support for EJB by the end of 1998. The remaining vendors plan to deliver EJB-compliant products in early 1999. The EJB API 1.0 specification defines what it means to support EJB. According to the specification, an EJB server must provide an execution environment for the enterprise bean, called an Enterprise JavaBeans container. An EJB container is also referred to as an EJB home.

Implicit Middleware Services

EJB provides implicit and automatic middleware services for any application component running within the EJB server. The EJB container is responsible for implementing the middleware services on behalf of one or more enterprise beans.

Bean-Managed versus Container-Managed Services

In all circumstances, an enterprise bean developer can choose to implement all middleware interactions directly within the bean application code. For example, an enterprise bean can manage its own transaction semantics by coding JTA transaction demarcation code within the application. This approach limits the reusability of the component since the transaction semantics cannot be changed without modifying the source code. For better portability and reusability, the developer should delegate middleware control to the container. If the container manages the middleware services, then the behavioral semantics are defined as attributes in a set of *deployment descriptor* objects. These values are defined at deployment time rather than development time.

EJB Services

The EJB container manages all lifecycle, naming, distribution, state, transaction, security, and (optionally) persistence services on behalf of the enterprise bean.

- **Lifecycle.** The EJB container is responsible for locating, creating, and destroying instances of an enterprise bean. The EJB container provides an *EJB Home* interface for each enterprise bean that implements the lifecycle services. By separating out the lifecycle services from the application code, the enterprise bean can automatically take advantage of any session pooling or load balancing services supplied by the EJB server.

- **Naming.** The EJB container automatically registers all EJB components in a naming or directory service. Clients retrieve references to the enterprise bean's EJB Home interface using JNDI, therefore the naming service could be implemented in any of a variety of naming or directory services.

- **Distribution.** The EJB container ensures that all enterprise beans are accessible through a remote interface called an *EJB Object* interface. The EJB Object interface is generated automatically by the EJB server at deployment time based on the remote interface definition provided with the enterprise bean (defined using RMI). The EJB Object interface provides a client view of the enterprise bean. The specification indicates that the EJB Object interface should support access through RMI and (optionally) CORBA. Other protocols could also be supported, such as an HTTP plug-in or DCOM. The EJB container uses the remote EJB Object interface to implement runtime services, such as state management, transactions, security, and persistence. The EJB container intercepts all method calls made through the remote interface and automatically performs the services before delegating the request to the enterprise bean.

- **State**. The EJB container manages the state of all active object instances. In a high-volume environment, numerous object instances need to share a limited amount of memory and resources. The EJB server may elect to swap an instance out of memory while it is not in use. The EJB container is responsible for saving and restoring the active object state each time the object is swapped in or out. The EJB container also must synchronize any cached database information with the database prior to any transaction commits or rollbacks.

- **Transactions.** An EJB server must support transactions. A transaction service ensures the integrity of a unit of work. A unit of work can consist of a single method call accessing a single data resource or it could incorporate a number of method calls, on multiple objects, manipulating any number of data resources. To ensure integrity, either all operations within the transaction complete successfully or all operations are reversed to their original form.

- **Transaction Management.** EJB supports automatic transaction management. An enterprise bean can elect to demarcate its own transactions using JTA or it can delegate transaction management to the EJB container. The EJB container manages the transaction context and automatically starts and commits transactions on behalf of the enterprise bean. For container-managed transactions, the transaction semantics are defined in a *deployment descriptor* object.

- **Transaction Coordination.** An EJB server must support distributed transaction coordination. The EJB server can provide its own transaction coordination service or it can delegate transaction coordination to an external transaction service provider, such as a database. The JPE JTS specification defines a standard Java transaction coordination service based on the CORBA Object Transaction Service (OTS). Although JTS is recommended, the EJB specification does not require a JTS-compliant transaction service. The transaction service used by the EJB server determines what kind of transactions are supported. A full implementation of JTS supports distributed, heterogeneous, two-phase commit transactional integrity against any number of transactional resource managers, such as databases, files, and message queues. JTS also defines a transaction interoperability protocol that allows transaction coordination with heterogeneous transaction environments, such as CORBA OTS or CICS/390. Most database transaction coordination services support only homogeneous transactions. For simple homogeneous environments, database-based transaction management is usually sufficient. For more complex systems, we strongly recommend using a complete implementation of JTS.

- **Security.** EJB supports automatic security checking using standard Java Security and access control lists (ACLs). ACLs are defined in a *security descriptor* object. The EJB container performs automatic security checking on each method call, matching the user identity or role to permissions defined in the ACL. The EJB server determines which authentication services can be used to establish user identity and what type of privacy and data integrity services are supported on all remote communications. For example, an application server may provide support for secure communications based on SSL or authentication based on X.509 certificates. We strongly recommend using SSL and X.509 certificates for any applications that support access from outside the firewall.

Optional Persistence Services

The EJB 1.0 specification defines, but does not require, automatic persistence services. EJB objects can be persisted in any permanent data store, such as a relational database, an object database, or a file system. The EJB specification does not dictate what mechanisms can be used to implement persistence. The JPE JDBC API provides a standard low-level SQL-based API to access relational databases. The American National Standards Institute (ANSI) is in the process of defining a standard embeddable SQL binding for Java called SQLJ. The Object Database Management Group (ODMG) has defined a standard Java binding to access object databases and object/relational mapping services. Although the service is optional, we view automatic persistence as an extremely desirable feature in an EJB application server. Automatic persistence provides a clean separation between the Java objects and the underlying data store. Therefore an enterprise bean could transparently take advantage of any persistence mechanism supplied by the application server.

Session versus Entity Beans

EJB defines two different types of enterprise beans. Session beans represent transient objects, and entity beans represent persistent objects. Entity beans are not required for EJB 1.0 compliance, and automatic persistence services are only available with entity beans.

Session Beans

Session beans are transient objects that exist (generally) for the duration of a single user session. A session bean represents work in progress on behalf of a single client. Referring back to Illustration 2 (see page 10), the Teller bean would be implemented as a session bean. The Teller bean represents the work being performed by the user for the duration of the user session, but it does not represent any persistent data.

Accessing Persistent Data

Session beans often make use of persistent information that is maintained in a database. An initialization routine can be set up for a session bean to automatically retrieve the appropriate data when the object is created. During the course of the session, the user may manipulate the data within the object, and any modifications to the data within the object must be manually written to the database before the object is destroyed. If the system crashes during processing, all data in the session bean will be lost.

Stateless and Stateful Session Beans

A session bean can be stateless or stateful. A stateless session bean does not maintain any user state between method calls; therefore, any instance of the bean can service any method request. A stateful bean maintains some user information across method requests; therefore, a user must continue to use the same bean instance throughout the user session. The EJB container maintains the session state on behalf of the bean. Historically, most business-critical, high-volume transaction systems have relied exclusively on stateless procedures. Stateless procedures can be pooled and recycled in the same fashion as threads and database connections to support faster response time and better scalability. Using the same technique, an EJB server can maintain a pool of reusable stateless session bean instances. Stateless beans support easier replication, fault recovery, and system failover. But stateless operations don't necessarily represent the real world. Many operations consist of multiple stages, and it can often be much more convenient and efficient to maintain state across method calls than to recreate state for each method call. Stateless objects contradict the essence of object-oriented processing, which marries code to data. With the increased adoption of object-oriented technology in enterprise application development, there is a significant push to use stateful and persistent objects.

Entity Beans

Entity beans are persistent objects that exist (generally) for an extended period of time. An entity bean provides an object abstraction for data stored in a database or some other persistent data store. Again referring back to Illustration 2 (see page 10), the Account bean would be implemented as an entity bean. The Account bean represents an object that is stored in the database. The account object could be stored as a row within a relational table, an object in an object database, or a file within a file hierarchy. When a user creates an entity bean, a new object is stored in the permanent data store. When a user later requests use of the entity bean, the object is retrieved from the data store and activated in memory. Any modifications made to the object are saved in the data store. Each entity bean has a unique identifier, and the entity bean is recoverable following a system crash. Multiple users can access a single entity bean instance.

Container-Managed Persistence

An entity bean can manage its own persistence or delegate persistence to its container. If the bean delegates persistence to the container, the container automatically performs all data retrieval and storage operations on behalf of the bean. Persistence semantics are defined in a *deployment descriptor* object.

Benefits of Container-Managed Persistence

Container-managed persistence simplifies enterprise bean development. Developers do not need to code any JDBC, SQLJ, or ODMG code within the application. Container-managed persistence also provides better support for portability and enables more flexible deployment options. If the enterprise bean manages its own persistence, the chances are much higher that bean will require a specific database for deployment.

Pluggable Persistence Classes

Container-managed persistence completely separates the entity bean from its persistence mechanism. The persistence mechanism is identified and mapped at deployment time. In fact, the persistence mechanism can change simply by redeploying the component and specifying new persistence semantics. Therefore the object can be shifted—from one relational database to another relational database, from a relational database to an object database, or to some other data store—without modifying the component application code.

Comprehensive support for Enterprise APIs

JPE API Requirements

An EJB environment makes extensive use of distributed infrastructure services, and therefore EJB relies on many of the other JPE APIs. JNDI, RMI, and JTA are required by the EJB 1.0 specification.

- **JNDI.** The EJB container automatically registers an enterprise bean in a naming or directory service. Although the container could use a proprietary interface to register the object, the specification assumes the use of JNDI. Client applications in turn use JNDI to obtain a reference to the object. JNDI support is now available for most naming and directory services. If an application server supports JNDI, the application server can use any directory service, including one that is already installed and in use.
- **RMI.** All enterprise beans support remote access, and remote interfaces are implemented using RMI. RMI is the preferred Java client interface because it is the simplest. RMI provides a completely native programming interface for Java clients. RMI also provides a very powerful and dynamic remote interface that allows any client to obtain access to the object at any time. Using RMI's pass-by-value capability, a client can download a client proxy object at runtime. EJB remote interfaces can also be implemented using CORBA and Java IDL. The Java IDL interface provides support for CORBA clients written in languages other than Java. Other EJB client interfaces can also be generated to support protocols, such as DCOM or HTTP plug-ins.
- **JTA**. The Enterprise JavaBeans model supports distributed transactions. EJB applications must use JTA to demarcate transactions and to manage the transaction context. JTA provides a high-level transaction interface that is compatible with a variety of transaction management services.

Other JPE APIs

The other JPE APIs are not required by the EJB specification, but a standards-focused EJB Java application server should include support for Servlets, JTS, JDBC, JMS, and Java IDL. Some application server vendors may use proprietary APIs in place of the standard JPE APIs. Using proprietary APIs within enterprise beans will severely compromise the portability and reusability of the application components.

Client Support

Java RMI

Every enterprise bean must define a remote interface using RMI. The EJB container exposes the remote interface to users. When a client invokes a method, the EJB container intercepts the request, implements services, and delegates the request to the enterprise bean.

CORBA IDL

In addition to RMI, the EJB container can generate a CORBA IDL interface from the remote interface definition, adding support for CORBA clients in C++, Smalltalk, and other languages.

ActiveX Clients

The EJB specification does not mention ActiveX clients, but an RMI interface can be wrapped in a COM object to support access from popular Windows development languages, such as Visual Basic or PowerBuilder.

HTML Client APIs

Java Servlets and Java Server Pages (JSPs) provide the standard JPE APIs to support HTML clients. Servlets and Java Server Pages (JSPs) interact with EJB components through the standard RMI interface. Servlets and JSPs are Java components that run on a Web server. Browser clients accessed servlets and JSPs by specifying the appropriate URL through HTTP. The Servlets and JSPs provide automatic HTTP session management, and they dynamically generate HTML. Servlets run as standard server extensions. JSPs enable direct integration of Java code with static HTML content through a set of standard HTML tags. Servlet and JSP support can be installed on any Web server that supports Java. Plug-in support for servlets and JSPs is available for Web servers from Sun, Microsoft, Netscape, IBM, and Apache. Most Java application servers provide support for HTML clients, although in many cases, HTML support is implemented using proprietary APIs based on CGI, NSAPI, or ISAPI Web server extensions rather than Servlets of JSPs.

The Power of Choice

Java Platform Independence

By default, a Java application server must support the execution of server-side Java applications or components; therefore a Java application server must provide (or provide access to) a Java runtime environment (JRE). Some Java application servers provide their own JREs. Others make use of the more popular JREs that already exist. An embedded JRE provides consistency and potentially added value across platforms, but it makes the customer dependent on a single vendor to implement upgrades on all platforms. It is highly desirable if a Java application server supports a certified Java-compatible runtime environment, designated by the steaming coffee cup logo. A Java-compatible runtime environment will support any 100% Pure Java object. At the same time, it is highly desirable to take advantage of any special features made available on a specific platform. For example, the Microsoft JRE (JView) provides exceptional integration with COM. A Java application server on the Windows platforms should be able to exploit the capabilities of JView. For the most flexibility, a Java application server should let the customer determine which JRE to use.

Tool Independence

A number of application servers provide integrated application development tools. In some cases, the provided development tool is the only tool that can be used to develop applications for the server. While these environments often offer tremendous productivity enhancements and specialized features, they greatly limit the use of third-party application components or packaged application features. On general principle, we prefer an open tools approach. Most application servers support Java development using any Java development tool.

Database Independence

The EJB specification does not dictate how objects access databases or how objects should be persisted. Many Java application servers do not supply integrated persistence services. Others rely on proprietary database access routines. Still others rely on JDBC. A few application servers provide integrated persistence service based on an embedded database (object or relational) or an integrated object/relational mapping service. We generally recommend taking advantage of automatic persistence services when available, with one caveat. Organizations must weigh the trade-off of locking themselves into a single built-in service versus being able to choose a persistence technology. We expect there will be a lot of innovation in object persistence technologies within the next few years, so choosing a solution that supports only one persistence mechanism will limit future options.

Integration Services

EJB application systems will not operate in a vacuum. An EJB server should provide mechanisms to allow enterprise beans to interoperate with other environments, such as CORBA, COM, and legacy application systems.

EJB Scalability and Reliability

Distribution and Resource Recycling

Application servers enhance the scalability of an environment by automatically distributing processing load across multiple processors and by effectively pooling and recycling scarce system resources. At the same time, an EJB server should simplify and automate the process of building distributed object applications. Using an application server, a Java developer should be able to build better applications with less code. For example, a Java application server should transparently manage session pooling, multithreading, active object state, and database connection pooling.

Connection, Session, and Thread Pooling

An EJB server is responsible for managing the lifecycle of enterprise bean objects. The EJB container creates and destroys objects on behalf of clients, establishes network connections and communication sessions, and dispatches object requests. The rate at which an application server can create new objects and allocate connections, sessions, and threads will have a tremendous impact on the overall performance and scalability of the system. One of the most critical benchmarks to look for in an application server is the number of objects that can be created per minute. One way to increase scalability is to pre-establish a pool of network connections, communication sessions, and execution threads that can be allocated to requests as needed. As users release these resources, they are recycled and placed back in the pool for the next request.

Load Balancing

Some high-end application servers support highly sophisticated dispatching services to increase the scalability and reliability of the system. For example, a distributed application server with TP monitor-type services can support object replication and load balancing. Multiple copies of an object can be deployed across a number of different processors, and requests can be transparently dispatched to any one of the object replicas to balance the load across all available processors. The load-balancing algorithms are likely to vary from application server to application server. For example, requests might be distributed based on a user identifier (such as a TCP/IP address) or on a parameter specified in the request. Or requests could be dispatched using a simple round-robin process. The more sophisticated systems can distribute requests based on actual system load or user-defined system rules.

Failover and Recovery

The object replication scheme also enables automatic and potentially transparent failover and object recovery. If an object fails for some reason, some application servers can automatically transfer the request to another object replica for processing.

Full State Replication

Replication and failover procedures are fairly straightforward when dealing with stateless servers. But the process can get more complicated when dealing with objects that maintain state. In order to support automatic and transparent failover of stateful objects, the application server must support fully synchronized replication of object state, not just object classes.

Watch for Single Point of Failure

Although many load balancing schemes support replication to increase the reliability of the application, in some cases the dispatcher can be a single point of failure. If the dispatcher goes down, then the entire application system might go with it.

Resource Management

Once an object has been created and activated, it consumes a set of scarce resources in the application server. During heavy usage, a single application server may not have sufficient resources to concurrently support all active objects in memory. Therefore an application server should have a mechanism to regain resources or to dynamically increase the resources available to support more objects.

Swapping Out Idle Objects

One mechanism that can be used to regain resources is to temporarily swap idle objects out of memory. Traditionally, application servers evict objects based on a least-recently-used algorithm. Before evicting an object, the EJB container passivates the object and stores its active state (the information that pertains to the specific user session). Then the next time the user invokes a method on the object, the EJB container restores the active state and reactivates the object. The speed at which an application server can activate and passivate objects from memory will have a significant impact on the overall performance of the application server.

Database Connection Pooling

A database connection is probably the most expensive resource used by an application. It takes longer to establish a connection than it does to execute a query. In addition, a modern relational database can only support approximately 300 concurrent connections; therefore it is critical to implement shared and pooled database connections in a large-scale application system. All Java application servers provide some level of pooled database connections, although performance characteristics may vary based on the efficiency of the connection allocation process.

Caching Data Access Results

In addition to database connections, it’s also a good idea to cache data access result sets. Some application servers provide automatic services that simplify the data access process and increase performance and capability. A top-of-the-line data caching system automatically transfers data updates in the cache to the database and automatically refreshes the cache when changes are made to the database. In some cases, an application server allows multiple users to share a single data cache.

EJB Management

Deployment Tools

A tremendous amount of information about an enterprise bean is defined at the time the bean is assembled with other beans and deployed as an application system in an EJB server. Much of the application’s behavior is defined as a set of attributes in a variety of deployment descriptor objects. Deployment descriptors are used to define the transaction semantics, security requirements, and persistence mapping information, among other things. The EJB server should provide a set of deployment tools and wizards that guide the deployer through the process.

Runtime Monitoring Tools

Once the components are deployed, other tools are required to monitor and manage the runtime environment. The monitoring tools should enable operations to quickly and easily identify any problems that might be occurring and make adjustments to the environment as needed. Although the application may be distributed, the monitoring tool should support management of the entire environment from a single console.

Administration Tools

Administration tools are used to manage users, security, and the EJB server environment. Administration tools would also be used to distribute new versions of system or application software, preferably with little or no physical contact with client machines.

Auditing and Control

In secure systems, the EJB server should maintain an audit trail for all sensitive operations. Any security violations or system exceptions should be captured for review and examination.

WebLogic Tengah

General-Purpose EJB Server

WebLogic is the first vendor to release a fully compliant EJB 1.0 server. WebLogic has been very involved in the definition of the EJB specification, and three previous versions of Tengah have supported preliminary implementations of EJB (releases 0.4, 0.5, and 0.8). Tengah is a general purpose Java application server that will suit the requirements of most organizations. WebLogic has been a premier provider of RMI and JDBC technologies. The Tengah application server combines WebLogic's highly scalable RMI framework and multi-tier JDBC Type III middleware services with support for server-side application components.

Pure Java Server

Tengah is implemented entirely in Java and can be deployed on any platform that supports Java. WebLogic officially supports the product on Windows 95, Windows NT, Netware, 0S/400, and the leading Unix platforms.

Complete EJB Implementation

Tengah is a complete implementation of EJB, providing full support for all optional features, including entity beans and container-managed persistence. Any EJB-compliant application component can be deployed in Tengah.

JPE API Support

Tengah also provides extensive support for the JPE APIs. Tengah provides client access through a highly scalable implementation of RMI. Web clients are supported through Servlets and JSPs. Enterprise beans can access relational databases using multi-tier JDBC and a full suite of native JDBC drivers. Tengah provides a powerful implementation of JNDI that supports a variety of directory services, including Novell NDS, Sun NIS+, Microsoft Active Directory, and any LDAP directory. Tengah manages transactions using JTA and JTS. And WebLogic is in the process of implementing full JMS support for an integrated Tengah Events service.

Tengah JTS Implementation

Tengah includes an implementation of JTS that supports distributed homogeneous transactions. Tengah JTS includes a lightweight two-phase commit protocol that allows distributed applications running on multiple servers to participate in a single transaction, updating multiple, distributed homogeneous databases. But Tengah does not provide an integrated transaction logging and recovery service. Instead, Tengah delegates the low-level transaction management services to an external database management system. Using the advanced transactional capabilities of Tengah/JDBC, Tengah can support multiple logical database connections through a single physical connection, thereby allowing the database to coordinate the distributed transaction.

Security

Tengah provides authorization services using Java Security and access control lists. Tengah supports authentication using a simple sign-on process, and it provides additional support for digital certificates. All network communications can be secured using SSL, HTTPS, or integrated RSA security.

Client Support

Tengah can support a variety of clients through an integrated remote access framework based on WebLogic's implementation of RMI. The remote access framework supports Web clients, desktop clients, and network appliances. Web browser users can access the environment through HTML and HTTP using a Servlet or a Java Server Page. Tengah supports Java clients through a standard RMI proxy, and it supports other language clients by wrapping the RMI proxy with ActiveX or CORBA.

Tengah COM Support

Tengah provides very tight integration with COM. If requested, the client generator can automatically wrap an EJB remote interface with an Active/X control. The Active/X control can be easily integrated with any COM-compliant application development tool or application, such as Visual Basic, PowerBuilder, Delphi, Active Server Pages, Internet Explorer, Word, or Excel.

Open Approach

Tengah uses an "open tools and technology" approach. Tengah applications can be developed using any Java development tool. Although Tengah supplies a default persistence mechanism based on JDBC, alternate persistence mechanisms are supported. Tengah applications can access any type of data source, including relational and object databases, flat files, and live datafeeds. Tengah provides built-in support for both COM and CORBA. Tengah can also support integration with legacy application systems.

Exploiting Resident JVMs

Tengah does not provide its own JVM. Instead, Tengah relies on the JVMs supplied by the platform vendors, fully exploiting any extensions provided. For example, Tengah on NT takes full advantage of the COM integration services provided in Microsoft JView.

Tengah Architecture

The Tengah application server provides a highly scalable execution environment. Illustration 3 shows an overview of the Tengah architecture.

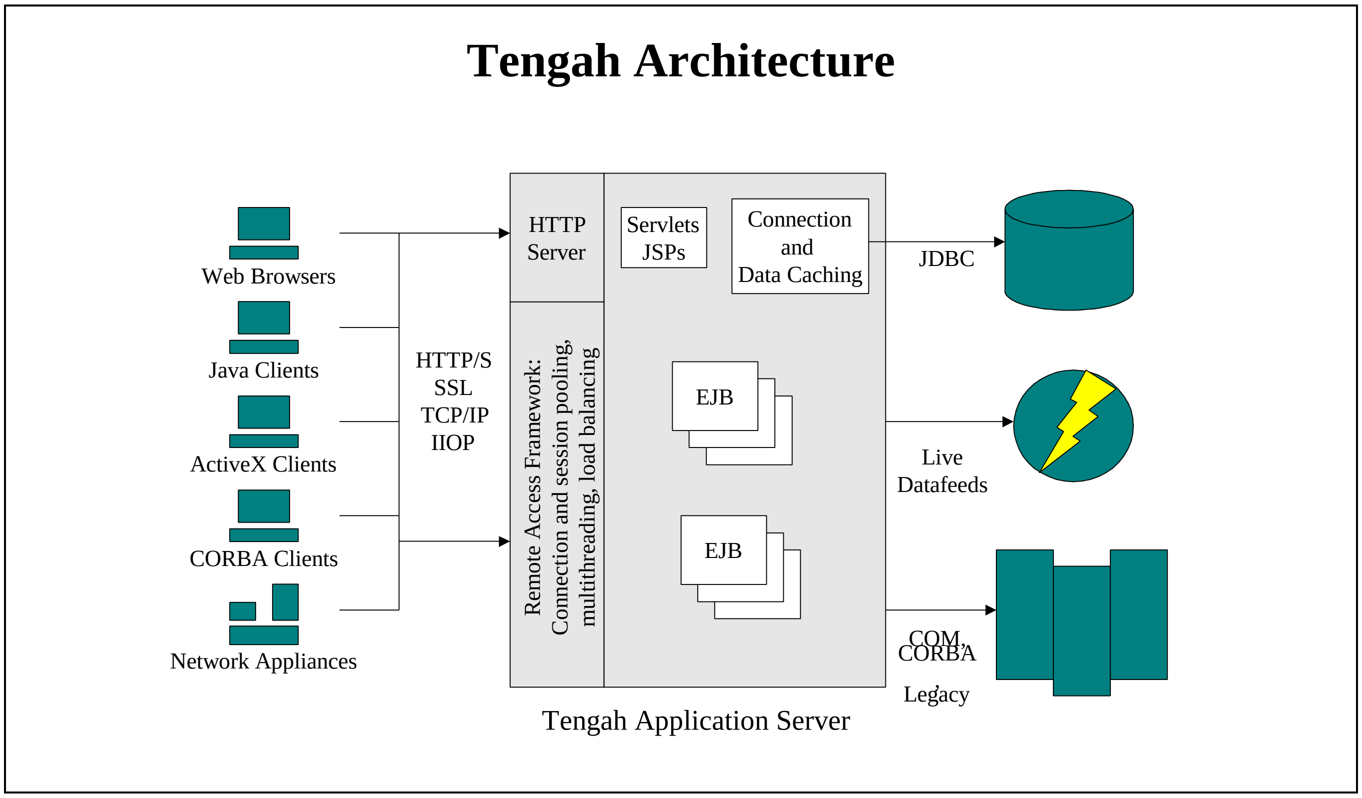

Illustration 3. Tengah supports various types of clients through its scalable remote access framework. Tengah supports a variety of network protocols, including TCP/IP, SSL, IIOP, HTTP, and HTTPS. Web clients access the environment through Servlets or Java Server Pages. The remote access framework manages network connection pooling, session pooling, multithreading, and load balancing. EJB components can access many types of data and integrate with other application systems through COM, CORBA, or custom connections. Tengah/JDBC provides comprehensive database connection pooling and data caching.

Highly Scalable Remote Access Framework

Tengah supports client access through a highly scalable remote access framework. The framework cleanly separates the programming interface from the underlying communications protocols. Currently, the remote access framework supports the RMI programming interface. RMI is a native Java remote access interface that, form the client’s point of view, makes the remote object behave like a local object. The EJB specification dictates that remote interfaces should be implemented in RMI. The Tengah remote access framework is a complete implementation of the Java RMI specification with a number of powerful extensions that support much greater scalability than Sun’s reference implementation of RMI. Sun RMI does not support session pooling or shared network connections. Tengah integrates RMI with the resource sharing services of an application server. Tengah automatically manages connection, session, and thread pooling for all RMI application requests.

Protocol Independence

The RMI API is a programming interface, not a complete middleware specification. RMI is designed to be protocol independent. Sun RMI currently supports only one communication protocol, JRMP, although JDK 1.2 will provide additional support for IIOP. WebLogic's RMI implementation is protocol independent, and requests can be passed over TCP/IP sockets, SSL, HTTP, HTTPS, or IIOP.

Load Balancing and Clustering

The Tengah remote access framework also supports application server clustering and load balancing. The built-in load balancing service can distribute requests across multiple EJB instances or across a cluster of Tengah servers. WebLogic provides a set of default load balancing policies, or users can define their own policies. The load balancing service is tightly integrated with both the remote access framework and Tengah JNDI. The load balancing policies are implemented directly into the RMI client proxy routines, called clustering stubs. When a client makes a request, the clustering stub queries JNDI to locate all available servers, then executes the policies to select a specific server.

No Single Point of Failure

Unlike most other application servers, Tengah's load balancing system is not dependent on a single dispatcher service. Most application servers implement a load balancing service as an HTTP plug-in. This approach exposes the environment to a single point of failure. The HTTP plug-in also only supports Web clients. Tengah load balancing occurs within the client application, supporting Web clients, desktop clients, and network appliances.

Recovery and Failover

The clustering stub also provides an added benefit of supporting automatic recovery and failover. If the client loses contact with its server, the clustering stub can automatically redirect the request to an alternate server.

Database Connection Pooling and Data Caching

Tengah uses its top-of-the-line JDBC Type III drivers to provide the highest quality database access services. Tengah/JDBC pre-establishes a set of database connections, which are allocated to objects as required. Multiple objects can share a single physical database connection, dramatically increasing the scalability capabilities of the database. Tengah also caches data result sets to increase performance and reduce I/O operations.

Management Tools

Tengah provides a full suite of management tools to simplify Tengah administration. A deployment wizard guides developers through the process of defining components to the Tengah environment. A graphical management console provides operations with a complete view of the runtime environment. Application components can be dynamically replicated or relocated from the console. Tengah is fully instrumented to collect runtime statistics, diagnostics, and security audit information.

Zero Administration Client

Tengah provides an automated push mechanism to distribute and install software on any client system. The Zero Administration Client (ZAC) is a small piece of software that must be installed on the client system. Once installed, ZAC can automatically manage the installation of new applets, applications, program libraries, or system software on the client without user intervention.

The Bottom Line

EJB is Now Available

Tengah is the first generally available implementation of EJB 1.0 and provides testament that the technology works. Additional implementations are now also available from a number of other vendors, and more implementations are in development. EJB has been universally adopted by the Java application server community, and the application vendors are beginning to adopt the component development approach.

Standard Environment

Tengah rigidly conforms to the EJB specification and fully supports all required and optional features, including entity beans and container-managed persistence. Any EJB-compliant application component can be deployed in Tengah. Tengah also supports the JPE APIs, including Servlets, RMI, JDBC, JTA, JTS, and JNDI.

Open Approach

Tengah doesn’t dictate the use of any specific clients, development tools, platforms, protocols, directories, or databases. Developers and administrators have complete freedom of choice when configuring the system. Tengah easily adapts to use whatever systems are already in place.

Highly Scalable, Robust Architecture

Tengah uses a highly scalable, robust architecture to support Java server components. Tengah’s unique approach implements much of the application server functionality directly into the RMI communications system. RMI is the native communications infrastructure for Enterprise JavaBeans, but the standard Sun implementation doesn’t support the kind of performance and scalability characteristics required by a high-end application server. WebLogic has implemented a number of powerful extensions to RMI to make the system much more scalable and reliable.

EJB Advantages

We strongly recommend a move to EJB and component-based computing. EJB dramatically simplifies the process of developing high-volume distributed application systems. EJB is especially well suited for Web applications and e-commerce systems. EJB automates the complex infrastructure programming so the developers can concentrate on writing business logic. EJB increases developer productivity and improves application quality. Best of all, EJB enables you to deploy new applications more quickly.

Plug-and-Play Application Assembly

In addition to increased developer productivity, EJB supports plug-and-play application assembly. Extending Java's "write once, run anywhere" theme, EJB components can be developed using any Java development tool and can be deployed in any EJB application server. Components from any number of vendors can be assembled into working applications. Although the market for reusable server components is still quite small, the field is growing. The vendor community has resoundingly endorsed the technology.

Practical EJB Server

But to use these reusable components, you must select an EJB application server. Tengah provides a practical EJB application server that meets the requirements of all but the most rigorous application systems. Tengah is designed specifically to support component-based computing. Unlike many of its competitors, Tengah doesn't carry a lot of proprietary baggage, such as HTTP plug-in client interfaces, specialized class libraries, or integrated development tools. Tengah relies on standard Java interfaces and APIs. Tengah provides an excellent host for "write once, run anywhere" and plug-and-play application assembly.

For more information regarding WebLogic or to download an evaluation copy of Tengah, please go to http://www.weblogic.com, or call 1.800.WEBLOGIC.