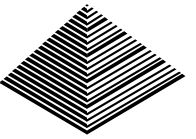
16



J2EE/EJB

A Case Study of an Industry-Standard Computing Infrastructure

with Anna Liu

Write Once, Run Everywhere
—Sun Microsystems's mantra for Java

Write Once, Test Everywhere
—Cynical Java programmers

This chapter presents an overview of Sun Microsystems's Java 2 Enterprise Edition (J2EE) architecture specification, as well as an important portion of that specification, Enterprise JavaBeans (EJB). J2EE provides a standard description of how distributed object-oriented programs written in Java should be designed and developed and how the various Java components can communicate and interact. EJB describes a server-side component-based programming model. Taken as a whole, J2EE also describes various enterprise-wide services, including naming, transactions, component life cycle, and persistence, and how these services should be uniformly provided and accessed. Finally, it describes how vendors need to provide infrastructure services for application builders so that, as long as conformance to the standard is achieved, the resultant application will be portable to all J2EE platforms.

J2EE/EJB is one approach to building distributed object-oriented systems. There are, of course, others. People have been building distributed object-oriented systems using the Object Management Group's (OMG) Common Object Request

Note: Anna Liu is a senior research engineer at the Software Architecture and Technologies Group, CSIRO, Sydney, Australia. She is also an adjunct senior academic at the University of Sydney.

Broker Architecture (CORBA) during the last decade. In the CORBA model, an object request broker (ORB) allows objects to publish their interfaces and allows client programs (and perhaps other objects) to locate these remote objects anywhere on the computer network and to request services from them. Microsoft, too, has a technology, .NET, for building distributed systems. The .NET architecture has similar provisions for building distributed object systems for Windows-based platforms.

We will start the chapter by looking at the business drivers that led to the creation of an industry standard architecture for distributed systems. Then we will discuss how the J2EE/EJB architecture addresses such needs. We will look at the typical quality requirements of Web-based applications and see how the J2EE/EJB architecture fulfills them.

16.1 Relationship to the Architecture Business Cycle

In the 1980s, the price/performance ratio for personal computers was gradually dovetailing with that of high-end workstations and "servers." This newly available computing power and fast network technology enabled the widespread use of distributed computing.

However, rival computer vendors kept producing competing hardware, operating systems, and network protocols. To an end-user organization, such product differentiation presented problems in distributed computing. Typically, organizations invested in a variety of computing platforms and had difficulty building distributed systems on top of such a heterogeneous environment.

The Object Management Group's Common Object Request Broker Architecture was developed in the early 1990s to counter this problem. The CORBA model provided a standard software platform on which distributed objects could communicate and interact with each other seamlessly and transparently. In this case, an ORB allows objects to publish their interfaces, and it allows client programs to locate them anywhere on the computer network and to request services from them.

However, CORBA was not the only viable distributed object technology for very long. Sun Microsystems soon pushed the Java programming language, which supports remote method invocation (RMI) and so, in effect, builds Java-specific object request broker functionality into every Java Virtual Machine (JVM). Java has the appeal of portability. Once a Java application is developed, its code is portable across all JVMs, which have implementations on most major hardware platforms.

Sun Microsystems did not stop with Java. J2EE was developed in the late 1990s using Java RMI as an underlying communication infrastructure. It became an industry-standard specification for the software community to more easily build distributed object systems using the Java programming language. J2EE soon

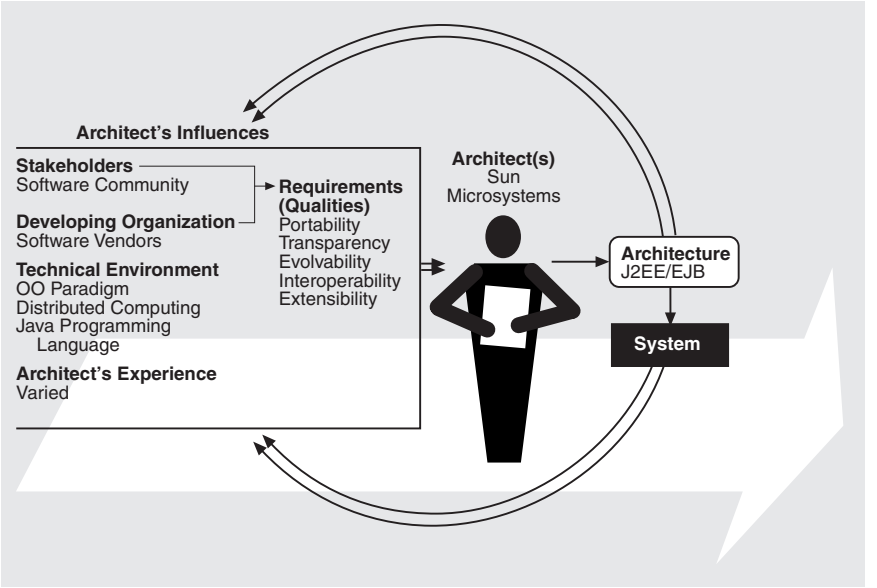


FIGURE 16.1 The ABC as it pertains to Sun Microsystems and J2EE/EJB

gathered momentum as software vendors rushed to implement it; Java programmers around the world showed great enthusiasm in developing e-commerce applications in "Internet time" using the J2EE framework. J2EE thus competed directly against CORBA as well as against the proprietary Microsoft technologies.

The ABC for J2EE/EJB is shown in Figure 16.1.

16.2 Requirements and Qualities

What are some of the goals of Sun Microsystems in developing the J2EE/EJB specification? How are these goals reflected in the qualities of the J2EE/EJB architecture?

THE WEB AND J2EE

In response to the increasing demands of Internet-enabled business systems, more and more enterprise information systems are constructed using distributed object technology. These systems require scalability, high performance, portability, and security. They need to handle large volumes of requests generated by the Internet community and must be able to respond to these requests in a timely fashion.

For many e-business organizations, the most challenging thing right now is a successful Web site. Successful sites attract large volumes of hits, and large volumes of hits stress the site software, as mentioned in Chapter 13. On the Internet, it is not uncommon for sites to receive millions or many millions of accesses daily. Such numbers might not be too frightening if user requests are spread out evenly during the day, but this is often not the case. Requests often arrive in bursts, which place greater demands on Web site software.

In fact, industry folklore is rife with stories of e-business sites failing under unexpected client surges. For example, the Wimbledon Tennis tournament experienced almost 1 billion Web accesses in 1999, with 420,000 hits per minute (7,000 per second) during one match. Bear in mind, that the Internet is currently used by only a small portion of the globe's population; things have just started.

In this sense, then, the Internet has forever changed the requirements for enterprise software systems. The very nature of the Internet brings new pressures to bear on applications that are not commonly experienced by traditional networked information systems. The impact of quality attribute requirements, such as manageability, scalability, security, and availability, are radically increased when applications are exposed to potentially limitless numbers of concurrent users. Table 16.1 describes the quality requirements that any Web-based application must fulfill.

Sun Microsystems, in developing J2EE, aimed to provide a basis for technology that supports the construction of such systems. In particular, as part of the J2EE specification, EJB aims to

- provide a component-based architecture for building distributed object-oriented business applications in Java. EJBs make it possible to build distributed applications by combining components developed with tools from different vendors.
- make it easier to write applications. Application developers do not have to deal with low-level details of transaction and state management, multithreading, and resource pooling.

TABLE 16.1 Typical Web-Based Application Quality Attribute Requirements

Quality	Requirement
Scalability	System should support variations in load without human intervention
Availability/ Reliability	System should provide 24/7 availability with very small downtime periods
Security	System should authenticate users and protect against unauthorized access to data
Usability	Different users should be able to access different content in different forms
Performance	Users should be provided with responsive systems

More specifically, the EJB architecture does the following:

- Addresses the development, deployment, and runtime aspects of an enterprise application's life cycle
- Defines the contracts that enable tools from multiple vendors to develop and deploy components that can interoperate at runtime
- Interoperates with other Java APIs
- Provides interoperability between enterprise beans and non-Java applications
- Interoperates with CORBA

J2EE makes it possible to re-use Java components in a server-side infrastructure. With appropriate component assembly and deployment tools, the aim is to bring the ease of programming associated with GUI-builder tools (like Visual Basic) to building server applications. And, by providing a standard framework for J2EE products based on a single language (Java), J2EE component-based solutions are, in theory at least, product independent and portable between the J2EE platforms provided by various vendors.

Thus, in addition to the core requirements given in Table 16.1, Sun added a set of requirements that address the activities of a programming team. These additional quality attribute requirements are listed in Table 16.2.

TABLE 16.2 Sun's Quality Attribute Requirements for J2EE

Quality Attribute	Requirement
Portability	J2EE should be able to be implemented with minimal work on a variety of computing platforms
Buildability	Application developers should be provided with facilities to manage common services such as transactions, name services, and security
Balanced Specificity	Detailed enough to provide meaningful standard for component developers, vendors, and integrators, but general enough to allow vendor-specific features and optimizations
Implementation Transparency	Provide complete transparency of implementation details so that client programs can be independent of object implementation details (server-side component location, operating system, vendor, etc.)
Interoperability	Support interoperation of server-side components implemented on different vendor implementations; allow bridges for interoperability of the J2EE platform to other technologies such as CORBA and Microsoft component technology
Evolvability	Allow developers to incrementally adopt different technologies
Extensibility	Allow incorporation of relevant new technologies as they are developed

16.3 Architectural Solution

Sun Microsystem's approach to satisfying the quality attributes discussed in the previous section is through the specification of two major architectures: J2EE and the EJB. J2EE describes the overall multi-tier architecture for designing, developing, and deploying component-based, enterprise-wide applications. EJB is a key part of J2EE technology, reflecting the deeper technical requirements of buildability, extensibility, and interoperability. Both J2EE and EJB reflect balanced specificity—that is, the ability for competitors to develop differentiation on the offerings while building them on a common base.

The major features of the J2EE platform are

- A multi-tiered distributed application model
- A server-side component model
- Built-in transaction control

A simple deployment view of the J2EE multi-tier model is given in Figure 16.2. The elements of this architecture are further described in Table 16.3.

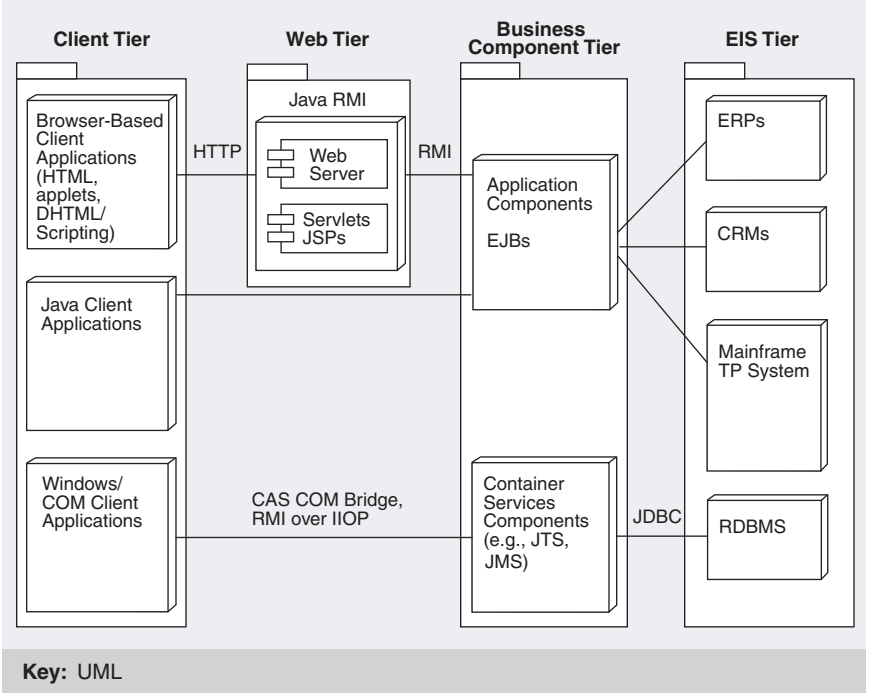


FIGURE 16.2 Deployment view of the J2EE multi-tier architecture

TABLE 16.3 Summary of J2EE Technology Components and Services

Component/Service	Description
Enterprise JavaBeans (EJB) Architecture	Specification defines an API that allows developers to create, deploy, and manage enterprise-strength server-side component-based applications
JavaServer Pages (JSP)	Provides a method for creating dynamic Web content
Java Servlet	Provides Web application developers with a mechanism for extending the functionality of a Web server
Java Messaging Service (JMS)	Provides J2EE applications with support for asynchronous messaging using either point-to-point (one-to-one) or publish-subscribe (many to many) styles of interaction; messages can be configured to have various qualities of service associated with them, ranging from best effort to transactional
Java Naming and Directory Interface (JNDI)	J2EE's directory service allows Java client and Web-tier servlets to retrieve references to user-defined objects such as EJBs and environment entries (e.g., location of a JDBC driver)
Java Transaction Service (JTS)	Makes it possible for EJBs and their clients to participate in transactions; updates can be made to a number of beans in an application, and JTS makes sure all changes commit or abort at the end of the transaction; relies on JDBC-2 drivers for support of the XA protocol and hence the ability to perform distributed transactions with one or more resource managers
J2EE Connector Architecture (JCA)	Defines a standard architecture for connecting the J2EE platform to heterogeneous Enterprise Information Systems, including packaged applications such as Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) systems
Client Access Services COM Bridge	Allows integration between COM and J2EE applications across a network; allows access to J2EE server-side components by COM-enabled client applications
RMI over IIOP	Provides developers with an implementation of Java RMI API over the OMG's industry-standard Internet Inter-ORB Protocol (IIOP); developers can write remote interfaces between clients and servers and implement them using Java technology and the Java RMI APIs
Java Database Connectivity (JDBC)	Provides programmers with a uniform interface to a wide range of relational databases and provides a common base on which higher-level tools and interfaces can be built

The role of each tier is as follows.

- *Client tier.* In a Web application, the client tier comprises an Internet browser that submits HTTP requests and downloads HTML pages from a Web server. In an application not deployed using a browser, standalone Java clients or applets can be used; these communicate directly with the business component tier. (See Chapter 17 for an example of using J2EE without a browser.)
- *Web tier.* The Web tier runs a Web server to handle client requests and responds to these requests by invoking J2EE servlets or JavaServer Pages

(JSPs). Servlets are invoked by the server depending on the type of user request. They query the business logic tier for the required information to satisfy the request and then format the information for return to the user via the server. JSPs are static HTML pages that contain snippets of servlet code. The code is invoked by the JSP mechanism and takes responsibility for formatting the dynamic portion of the page.

- *Business component tier.* The business components comprise the core business logic for the application. They are realized by EJBs (the software component model supported by J2EE). EJBs receive requests from servlets in the Web tier, satisfy them usually by accessing some data sources, and return the results to the servlet. EJB components are hosted by a J2EE environment known as the EJB container, which supplies a number of services to the EJBs it hosts including transaction and life-cycle management, state management, security, multi-threading, and resource pooling. EJBs simply specify the type of behavior they require from the container at runtime and then rely on the container to provide the services. This frees the application programmer from cluttering the business logic with code to handle system and environmental issues.
- *Enterprise information systems tier.* This typically consists of one or more databases and back-end applications like mainframes and other legacy systems, which EJBs must query to process requests. JDBC drivers are typically used for databases, which are most often Relational Database Management Systems (RDBMS).

THE EJB ARCHITECTURAL APPROACH

The remainder of this chapter focuses on the Enterprise JavaBeans architecture, which defines a standard programming model for constructing distributed object-oriented server-side Java applications. Because this programming model is standard, many beans that prepackage useful functionality can be (and have been) written. The EJB programmer's job is to bundle these packages with any application-specific functionality to create a complete application.

Not unlike J2EE, EJBs aim at realizing one of Java's major design principles—the oft-quoted "Write Once, Run Anywhere" mantra. The JVM allows a Java application to run on any operating system. However, server components require additional services that are not supplied directly by the JVM, such as transaction and security services. In J2EE and EJB, these services are supplied through a set of standard vendor-independent interfaces that provide access to the additional supporting infrastructure, which together form the services available in an application server.

A J2EE-compliant application server provides an EJB *container* to manage the execution of application components. In practical terms, a container provides an operating system process that hosts one or (usually) more EJB components.

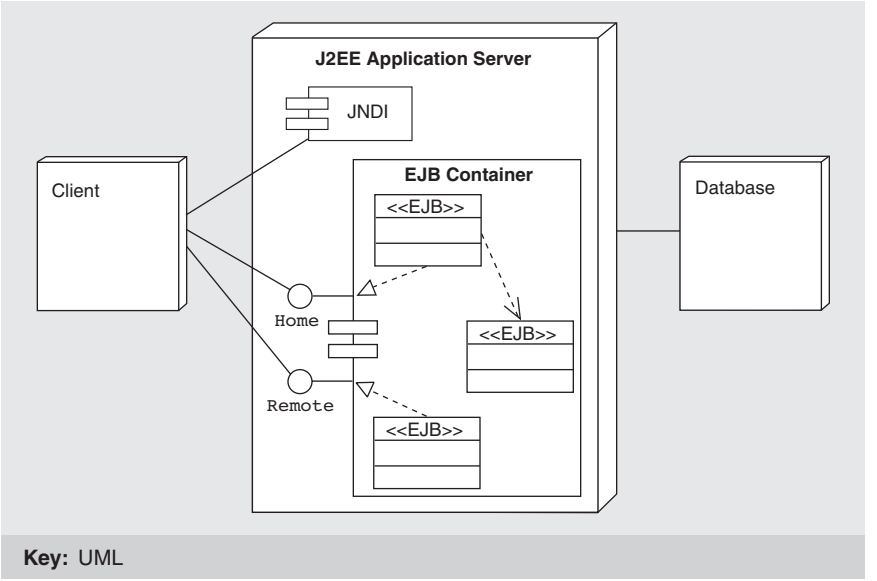


FIGURE 16.3 Example deployment view of the EJB architecture

Figure 16.3 shows the relationship between an application server, a container, and the services provided. In brief, when a client invokes a server component the container automatically allocates a thread and invokes an instance of the component. The container manages all resources on the component's behalf and manages all interactions between the component and the external systems.

The EJB component model defines the basic architecture of an EJB component, specifying the structure of its interfaces and the mechanisms by which it interacts with its container and other components. The model also provides guidelines for developing components that can work together to form a larger application.

The EJB version 1.1 specification defines two main types of components: *session beans* and *entity beans*.

- *Session beans* typically contain business logic and provide services for clients. The two types of session bean are known as *stateless* and *stateful*.
  - A *stateless session bean* is defined as not being *conversational* with respect to its calling process. This means that it does not keep any state information on behalf of any client. A client will get a reference to a stateless session bean in a container and can use it to make many calls on an instance of the bean. However, between each successive service invocation, a client is not guaranteed to bind to any particular stateless session bean instance. The EJB container delegates client calls to stateless session beans *as needed*, so the client can never be certain which bean instance it

will actually talk to. This makes it meaningless to store client-related state in a stateless session bean.
  - A *stateful session bean* is said to be conversational with respect to its calling process and therefore can maintain state information about the conversation. Once a client gets a reference to a stateful session bean, all subsequent calls to the bean using this reference are guaranteed to go to the same bean instance. The container creates a new, dedicated stateful session bean for each client that creates a bean instance. Thus, clients can store any state information they wish in the bean and can be assured that it will still be there the next time they access that bean. EJB containers assume responsibility for managing the life cycle of stateful session beans. The container writes out a bean's state to disk if it has not been used for a while and automatically restores the state when the client makes a subsequent call on the bean. This mechanism is known as *passivation and activation* of the stateful bean. We will discuss passivation in more detail later.

- *Entity beans* are typically used for representing business data objects. The data members in an entity bean map directly to some data items stored in an associated database. Entity beans are usually accessed by a session bean that provides business-level client services. There are two types of entity bean, *container-managed persistence* and *bean-managed persistence*. Persistence in this context refers to the way in which a bean's data (usually a row in a relational database table) is read and written.
  - With *container-managed persistence entity beans*, the data the bean represents is mapped automatically to the associated persistent data store (e.g., a database) by the container. The container is responsible for loading the data to the bean instance and writing changes back to the persistent data storage at appropriate times, such as the start and end of a transaction. Container-managed persistence relies on container-provided services and requires no application code—the container in fact generates the data access code so it is easy to implement.
  - With *bean-managed persistence entity beans*, the bean code itself is responsible for accessing the persistent data it represents, typically using handcrafted JDBC calls. Bean-managed persistence gives the bean developer the flexibility to perform persistence operations that are too complicated for the container or to use a data source not supported by the container—for example, a custom or legacy database. While bean-managed persistence requires more programmer effort to implement, it can sometimes provide opportunities to optimize data access and, in such cases, may provide better performance than container-managed persistence.

Table 16.4 summarizes how the EJB architecture supports Sun's key quality attribute requirements for the overall J2EE architecture. An example deployment view of the J2EE/EJB architecture is illustrated in Figure 16.4.

TABLE 16.4 How EJB Supports Sun's J2EE Quality Attribute Requirements

Goal	How Achieved	Tactics Used
Availability/ Reliability	J2EE-compliant systems provide ready-to-use transaction services that enhance availability and reliability of the application by providing built-in failure recovery mechanisms	Heartbeat Transactions Passive redundancy
Balanced Specificity	EJB services specified in terms of Java APIs, effectively defer implementation decisions to EJB application server implementers; detailed enough to provide a meaningful standard for component developers, vendors and integrators, but general enough to allow vendor-specific features and optimizations	Anticipate expected changes Abstract common services Hide information
Buildability	EJB application servers provide many ready-to-use services for building server-side Java applications, including transactions, persistence, threading, and resource management; developer is thus freed from low-level distribution details; Sun Microsystems provides a reference J2EE implementation; application server vendors also participate in the J2EE specification process	Abstract common services Maintain interfaces Hide information
Evolvability	Specification partitioned into separately evolvable subcategories; the Java Community Process coordinates Java specification requests and responses	Semantic coherence Hide information
Extensibility	Component-based approach to the EJB specification allows for future extensions; message-driven beans are a feature introduced in later versions of the EJB specification and workable with existing EJB systems; J2EE describes stable core technologies, such as EJB, JMS, JNDI, JTS, etc., needed by most component developers; over time, extensions, such as JCA, are gradually incorporated	Anticipate expected changes
Implementation Transparency	`Home` and `Remote` interface specifications encourage decoupling of interface specification and implementation. Implementation decisions can thus be deferred, and are transparent to the client; provide complete transparency of implementation details so that client programs can be independent of object implementation details (server-side component location, operating system, vendor, etc.)	Maintain existing interfaces Semantic coherence
Interoperability	Supports interoperation of server-side components implemented on different vendor implementations; also allow bridges for interoperability of the J2EE platform to other technologies such as CORBA and Microsoft component technology	Adherence to defined protocols
Performance	Distributed-component approach to J2EE/EJB allows performance tuning across multiple systems	Configuration files Load balancing Maintain multiple copies

(Continued)

TABLE 16.4 *Continued*

Goal	How Achieved	Tactics Used
Portability	Contracts between EJBs and containers ensure application components are portable across different EJB containers; J2EE describes roles for application component providers, assemblers, deployers, EJB server providers, EJB container providers, and system administrators, as well as precise contracts between various J2EE components and application components; application component (in theory) is thus portable across different J2EE containers; J2EE is based on a language that contains its own virtual machine and is available on most major platforms	Maintain existing interfaces Generalize modules Abstract common services
Scalability	J2EE multi-tiered architecture and component-based EJB architecture has built-in mechanisms for expanding the number of servers available in a configuration and to load balance among servers	Load balancing
Security	J2EE-compliant systems provide declarative, role-based security mechanisms and programmatic security mechanisms that are ready to use	Authentication Authorization Data confidentiality
Usability	J2EE-compliant systems provide Java technologies, such as JSP and servlets, that enable the rendering of content to suit different users	Separate user interface

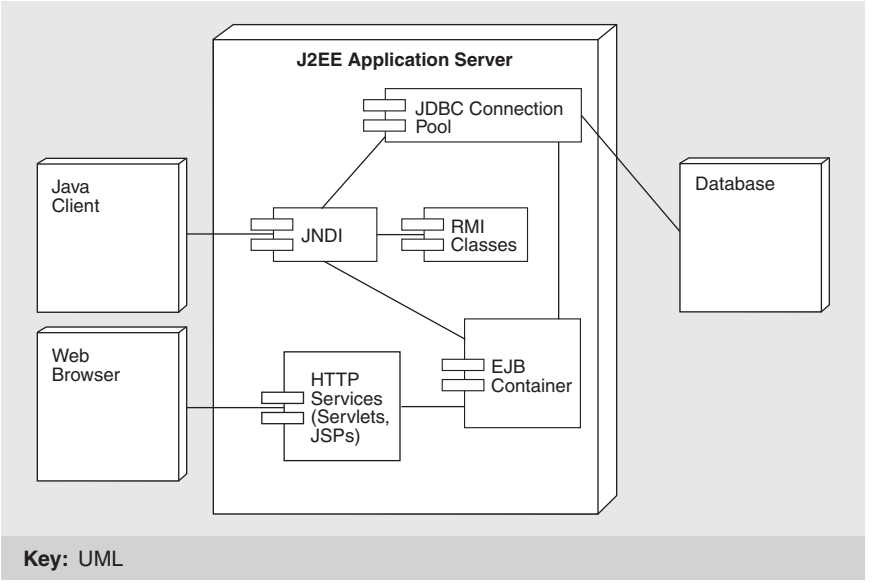


FIGURE 16.4 An example J2EE/EJB-compliant implementation

EJB PROGRAMMING

An EJB depends on its container for all external information. If an EJB needs to access a JDBC connection or another bean, it uses container services. Accessing the identity of its caller, obtaining a reference to itself, and accessing properties are all accomplished through container services. This is an example of an "intermediary" tactic. The bean interacts with its container through one of three mechanisms: callback methods, the `EJBContext` interface, and the Java Naming and Directory Interface (JNDI).

To create an EJB server-side component, the developer must provide two interfaces that define a bean's business methods, plus the actual bean implementation class. The two interfaces, `remote` and `home`, are shown in Figure 16.5. Clients use them to access a bean inside an EJB container. They expose the capabilities of the bean and provide all the methods needed to create the bean and update, interact with, or delete it.

The two interfaces have different purposes. `Home` contains the life-cycle methods of the EJB, which provide clients with services to create, destroy and find bean instances. In contrast, `remote` contains the business methods offered by the bean. These methods are application specific. To use them in the bean's `remote` interface, clients must use the bean's `home` interface to obtain a reference to the `remote` interface.

A simple `home` interface is shown in Figure 16.6. It must inherit from `EJBHome` and, in this example, contains a method to create an EJB of type `Broker`. Figure 16.7 shows the `remote` interface for the `Broker` EJB.

`Remote` interfaces must extend the `EJBObject` interface, which contains a number of methods that the container uses to manage an EJB's creation and life

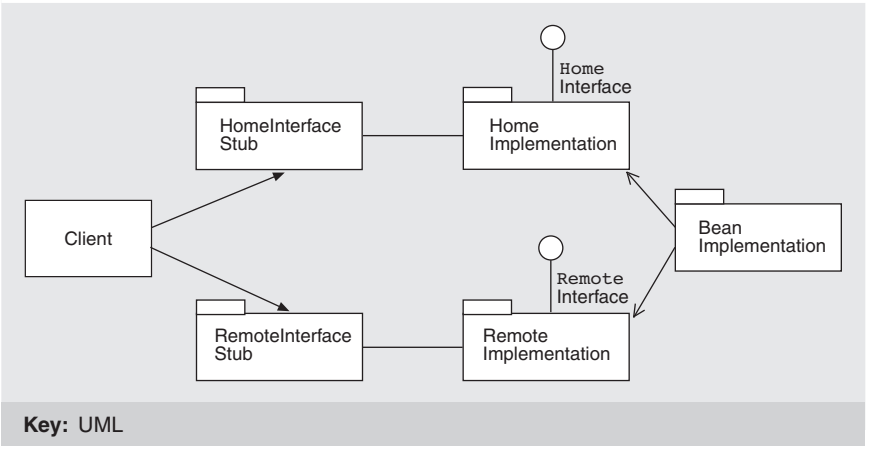


FIGURE 16.5 EJB package diagram

```
public interface BrokerHome extends EJBHome
    {
     /*
      * This method creates the EJB Object.
      *
      * @return The newly created EJB Object.
      */
     Broker create() throws RemoteException, CreateException;
    }
```

FIGURE 16.6 A simple `home` interface

```
public interface Broker extends EJBObject
{
 // Return the newly created account number
 public int newAccount(String sub_name, String sub_address, int
     sub_credit) throws RemoteException, SQLException;
 public QueryResult queryStockValueByID(int stock_id)
     throws RemoteException, SQLException;
 public void buyStock(int sub_accno, int stock_id, int amount)
     throws RemoteException, SQLException, TransDenyException;
 public void sellStock(int sub_accno, int stock_id, int amount)
     throws RemoteException, SQLException, TransDenyException;
 public void updateAccount(int sub_accno, int sub_credit)
     throws RemoteException, SQLException;
 public Vector getHoldingStatement(int sub_accno, int start_
     stock_id) throws RemoteException, SQLException;
}
```

FIGURE 16.7 The Broker `remote` interface

cycle. A programmer may wish to provide bean-specific behavior for the EJB, or may simply accept the default, inherited behavior. The client then uses `public` interfaces to create, manipulate, and remove beans from the EJB server. The implementation class, normally known as the bean class, is instantiated at runtime and becomes an accessible distributed object. Some sample client code, simplified, is shown in Figure 16.8.

EJB clients may be standalone applications, servlets, applets, or even other EJBs, as we will see shortly. All clients use the server bean's `home` interface to obtain a reference to an instance of the server bean. This reference is associated with the class type of the server bean's `remote` interface; so the client interacts with the server bean entirely through the methods defined in its `remote` interface.

In this next example, the `Broker` bean is acting as a stateless session bean that handles all client requests. Internally, it uses the services of a number of entity beans to perform the business logic. A sample of one of the `Broker` methods, `updateAccount`, is shown in Figure 16.9.

The `updateAccount` method uses an entity bean called `Account`, which encapsulates all of the detailed manipulation of the application's data—in this case, exactly how an account record is updated. The code in `updateAccount`

```
Broker broker = null;

// find the home interface
Object _h = ctx.lookup("EntityStock.BrokerHome");
BrokerHome home = (BrokerHome)
    javax.rmi.PortableRemoteObject.narrow(_h, BrokerHome.class);
// Use the home interface to create the Broker EJB Object
broker = home.create();
// execute requests at the broker EJB
broker.updateAccount(accountNo, 200000);
broker.buyStock(accountNo, stockID, 5000);

//we're finished...
broker.remove();
```

FIGURE 16.8 Simplified example EJB client code

```
public void updateAccount(int sub_accno, int sub_credit)
 throws RemoteException
{
 try {
     Account account = accountHome.findByPrimaryKey
        (new AccountPK(sub_accno));
     account.update(sub_credit);
 }
 catch (Exception e) {
     throw new RemoteException(e.toString());
 }
}
```

FIGURE 16.9 The `Broker` bean's `updateAccount` method

uses an entity bean `finder` method called `findByPrimaryKey`, which is provided by the `Account` bean in its `home` interface. This method takes the primary key for the account and accesses the underlying database. If an account record is found in the database with this primary key, the EJB container creates an `Account` entity bean. The entity bean methods—in this example `update`—can then be used to access the data in the account record. The `home` and `remote` interfaces for `Account` are shown in Figure 16.10.

The bean class for the entity bean implements the remote methods. The code for the `update` method is shown in Figure 16.11. It is very simple—in fact, a single line of executable Java code. This simplicity is due to the entity bean's use of *container-managed persistence*. The EJB container "knows" (we will see how soon) that there is a correspondence between the data members in the `Account` bean and the fields in an account table in the database the application is using.

Using this information, the container tools can generate the SQL queries needed to implement the `finder` method, and the queries needed to automatically read/write the data from/to the entity bean at the beginning/end of a transaction. In this example, at the end of the `Broker` session bean's `updateAccount`

```
public interface AccountHome extends EJBHome
{
 /*
  * This method creates the EJB Object.
  *
  * @param sub_name The name of the subscriber
  * @param sub_address The address of the subscriber
  * @param sub_credit The initial credit of the subscriber
  *
  * @return The newly created EJB Object.
  */
 public Account create(String sub_name, String sub_address,
     int sub_credit)  throws CreateException, RemoteException;
 /*
  * Finds an Account by its primary Key (Account ID)
  */
 public Account findByPrimaryKey(AccountPK key)
     throws FinderException, RemoteException;
}

public interface Account extends EJBObject
{
 public void update(int amount) throws RemoteException;
 public void deposit(int amount) throws RemoteException;
 public int withdraw(int amount) throws AccountException,
     RemoteException;
 // Getter/setter methods on Entity Bean fields
 public int getCredit() throws RemoteException;
 public String getSubName() throws RemoteException;
 public void setSubName(String name) throws RemoteException;
}
```

FIGURE 16.10 The `Account` bean's `home` and `remote` interfaces

```
public class AccountBean implements EntityBean
{
 // Container-managed state fields
 public int    sub_accno;
 public String sub_name;
 public String sub_address;
 public int    sub_credit;

 // lots missing ...
 public void update(int amount)
 {
     sub_credit = amount;
 }
}
```

FIGURE 16.11 The `Account` bean's `update` method

method, the data items in the `Account` entity bean are written back to the database, making the changes to the `sub_credit` field persistent. All of this is done without explicit control from the programmer, which contributes to the buildability of EJB-based systems.

DEPLOYMENT DESCRIPTORS

One of the major attractions of the EJB model is the way it achieves a separation of concerns between the business logic and the infrastructure code, an example of the "semantic coherence" tactic. This separation refers to the fact that EJBs are primarily concerned with pure business logic while the EJB container handles environmental and infrastructure issues such as transactions, bean life-cycle management, and security. This makes the bean components simpler—they are not littered with code to handle these additional complexities.

A bean tells the container which of the provided services it requires through a deployment descriptor. This is an XML document associated with an EJB. When a bean is deployed in a container, the container reads the deployment descriptor to find out how transactions, persistence (for entity beans), and access control should be handled. In this way the descriptor provides a declarative mechanism for how these issues are handled—an example of the "defer binding time" tactic.

The beauty of this mechanism is that the same EJB component can be deployed with different descriptors suited to different application environments. If security is an issue, the component can specify its access control needs. If security is not an issue, no access control is specified. In both cases the code in the EJB is identical.

A deployment descriptor has a predefined format that all EJB-compliant beans must use and that all EJB-compliant servers must know how to read. This format is specified in an XML Document Type Definition, or DTD. The deployment descriptor describes the type of bean (session or entity) and the classes used for `remote`, `home`, and the bean class. It also specifies the transactional attributes of every method in the bean, which security roles can access each method (access control), and whether persistence in the entity beans is handled automatically by the container or performed explicitly by the bean code.

The deployment descriptor for the `Broker` bean shown before is given in Figure 16.12. In addition to the attributes described, the deployment descriptor specifies that this is a stateless session bean and that a container-managed transaction is required to execute each of its methods (in the figure these attributes are in boldface for ease of reading). For example, if we simply change the `<session-type>` field in the XML to read `stateful`, the container will manage the bean very differently. Figure 16.13 shows the deployment descriptor for the `Account` entity bean. As well as the deployment attributes we have already seen, it tells the container the following:

- That it must manage persistence for beans of this type
- Where to find the JDBC data source for the database
- What primary key and data items must be mapped between the database and the entity bean

In Table 6.2, we presented Sun's quality attribute requirements for J2EE. In Table 16.5, we describe how some of these requirements are achieved by deployment descriptors.

```
<ejb-jar>
 <enterprise-beans>
     <session>
       <ejb-name>EntityStock.BrokerHome</ejb-name>
       <home>j2ee.entitystock.BrokerHome</home>
       <remote>j2ee.entitystock.Broker</remote>
       <ejb-class>j2ee.entitystock.BrokerBean</ejb-class>
       <session-type>Stateless</session-type>
       <transaction-type>Container</transaction-type>
     </session>
 </enterprise-beans>
 <assembly-descriptor>
     <container-transaction>
       <method>
          <ejb-name>EntityStock.BrokerHome</ejb-name>
          <method-intf>Remote</method-intf>
          <method-name>*</method-name>
       </method>
       <trans-attribute>Required</trans-attribute>
     </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

FIGURE 16.12 Deployment description for the `Broker` bean

TABLE 16.5 How Deployment Descriptors Support Sun's J2EE Quality Attribute Requirements

Goal	How Achieved	Tactics Used
Portability	Common code base can be developed for multiple target platforms; multiple versions of deployment descriptor can be configured at deployment time to suit different target platforms, making the developed application component portable across multiple target environments	Semantic coherence, generalize modules, configuration files
Buildability	Deployment descriptors enable separation of concerns: development of code and deployment configuration options	Semantic coherence, configuration files, generalize module
Balanced Specificity	Deployment descriptors in XML format, providing a meaningful standard format for encoding configuration options, but general enough for vendors to extend deployment descriptors with vendor-specific features	Configuration files, generalize module
Implementation Transparency	Details of deployment descriptor used by server-side components are transparent to the clients of the components	Use an intermediary

```
<ejb-jar>
 <enterprise-beans>
     <entity>
       <ejb-name>EntityStock.AccountHome</ejb-name>
       <home>j2ee.entitystock.AccountHome</home>
       <remote>j2ee.entitystock.Account</remote>
       <ejb-class>j2ee.entitystock.AccountBean</ejb-class>
       <persistence-type>Container</persistence-type>
       <prim-key-class>j2ee.entitystock.AccountPK</prim-key-class>
       <reentrant>False</reentrant>
       <cmp-field>
          <field-name>sub_accno</field-name>
       </cmp-field>
       <cmp-field>
          <field-name>sub_name</field-name>
       </cmp-field>
       <cmp-field>
          <field-name>sub_address</field-name>
       </cmp-field>
       <cmp-field>
          <field-name>sub_credit</field-name>
       </cmp-field>
       <resource-ref>
          <res-ref-name>jdbc/sqlStock_nkPool</res-ref-name>
          <res-type>javax.sql.DataSource</res-type>
          <res-auth>Container</res-auth>
       </resource-ref>
     </entity>
 </enterprise-beans>
 <assembly-descriptor>
     <container-transaction>
       <method>
          <ejb-name>EntityStock.AccountHome</ejb-name>
          <method-intf>Remote</method-intf>
          <method-name>*</method-name>
       </method>
       <trans-attribute>Required</trans-attribute>
     </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

FIGURE 16.13 Deployment description for the `Account` entity bean

16.4 System Deployment Decisions

What we have described thus far is J2EE/EJB as it was created by Sun. However, when deploying a J2EE/EJB system, there are a number of implementation issues that the architect needs to consider. The EJB component model is a powerful way to construct server-side applications. And although the interactions between the different parts of the code are a little daunting at first, with some exposure and experience with the model, it becomes relatively straightforward to construct EJB applications. Still, while code construction is not difficult, a number of complexities remain, including the following.

- The EJB model makes it possible to combine components in an application in many different architectural patterns. Which are the best, and what does "best" mean in a given application?
- The way beans interact with the container is complex and has a significant effect on the performance of an application. In the same vein, all EJB server containers are not equal—product selection and product-specific configuration are important aspects of the application development life cycle.

In this final section, we present some of the key design issues involved in architecting and constructing highly scalable EJB applications.

STATE MANAGEMENT—AN OLD DESIGN ISSUE IN A NEW CONTEXT

There are two service models that can be adopted in developing the EJB server tier—stateless and stateful models, implemented by stateless and stateful session beans.

We will take an online bookshop as an example. In the stateful version, an EJB can be used to remember customer details and to manage the items the customer is placing in an online shopping cart. Hence, the EJB stores the state associated with the customer's visit to the site. By maintaining this conversational state in the bean, the client is relieved from the responsibility of keeping track of it. The EJB monitors potential purchases and processes them in a batch when a confirmation method is invoked.

To make better use of limited system memory, stateful session beans are passivated when not used by the client, meaning that a bean's conversational state is written to secondary storage (typically disk) and its instance is removed from memory. The client's reference to the bean is not affected by passivation, but remains alive and usable. When the client invokes a method on a bean that is passivated, the container activates the bean by instantiating a new instance and populating its state with the information written to secondary storage.

This passivation strategy has great implications for scalability. If there is a requirement for large numbers of stateful session bean instances to service individual clients, passivation and activation may prove to be too high an overhead in terms of application performance.

Alternatively, a stateless session bean does not maintain conversational state on behalf of the client. The client must inform the server of session information, such as customer details and shopping cart contents, with each service request, because, for each request, the container may assign a different stateless session bean instance. This is only possible because of the pure stateless service model. Figure 16.14 shows usage of both stateful and stateless session beans.

To summarize, the advantages of stateless session beans include the following:

- There is no performance overhead in passivating and activating session beans that involve expensive disk reads and writes.

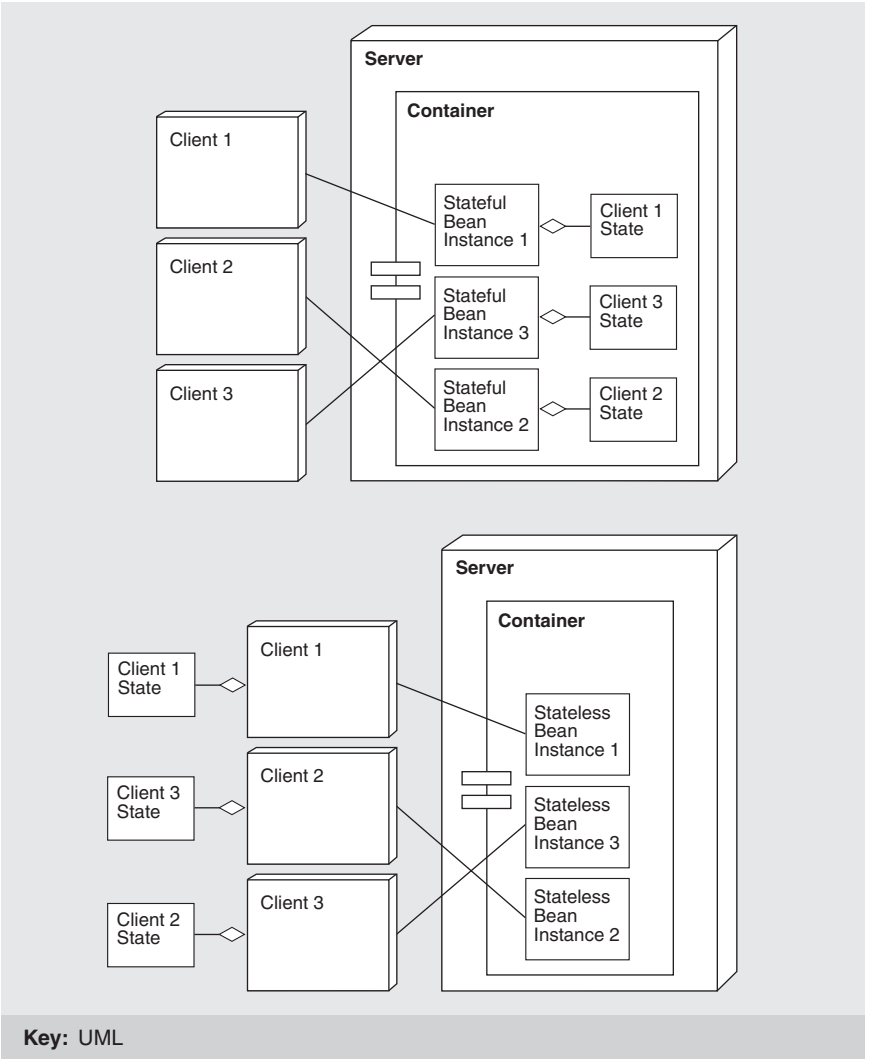


FIGURE 16.14 Clients' static bindings to stateful session bean instances and dynamic bindings to stateless session bean instances.

- Dynamic request routing means that requests can be routed to the least loaded server.
- If one session instance goes down, the request can be easily rerouted to another one.

The only disadvantage to the stateless approach is that more information needs to be passed between the client and the EJB on each request. Assuming that the

amount of data is not prohibitively large, the stateless session bean will most likely better support high system scalability.

Entity Beans—To Use or Not to Use? A common EJB design pattern is to provide a wrapper session bean that exposes services to the client and, at the same time, accesses the business data encapsulated in the entity bean to fulfill a client request. This represents a clean object-oriented programming model. Business data, usually represented in a relational format in a database, is now encapsulated in an object-oriented format (entity beans). The various `get` and `set` methods defined for entity beans make it easy for session beans to access this data. Additionally, if container-managed persistence is used for entity beans, the developer need not explicitly develop the database access code.

The risk here is a considerable performance penalty. Testing results show that, for a typical e-commerce system with an 85% read-only and 15% update transaction mix, the application architecture using entity beans achieves roughly half the system throughput compared to an architecture utilizing session beans only. The performance degradations have the following causes:

- The entity beans introduce an additional indirection layer rather than session beans directly accessing the business object in the database. Depending on which container implementation is used, the container may not automatically optimize calls to entity beans (from session beans) to a local call. In this case, the additional RMI call is expensive.
- The life-cycle management of entity beans in this additional layer can be expensive. Activation is equivalent to at least a single database/disk-read operation, and passivation is a database/disk-write operation.
- Additional beans participate in the transaction.

Of course, it is up to the application architect to decide if the benefits of entity beans outweigh the likely loss in system throughput.

DISTRIBUTION AND SCALING ISSUES

With the popularity of Web-enabled enterprise systems, businesses are finding their back-end systems unable to cope with the volume of incoming Internet traffic. There are two ways of increasing the processing power in the server tier:

- Scaling up, or "vertical" scaling, refers to the addition of computational and system resources—for example, adding memory to a single machine. This form of scaling relies on the application server having no inherent bottlenecks in its internal architecture. If this is the case, given more system resources and processor power, the application server software should be able to fully utilize the additional resources and increase system throughput as a result.

- Scaling out, or "horizontal" scaling, means that, instead of replacing an existing machine with a more powerful model, the server application is distributed across more than one machine. This should increase overall system resources and processing power by making additional machines available to the application.

Scaling out is usually regarded as more difficult to implement than scaling up, because it requires more complex configuration and system management. The application server must also provide load-balancing mechanisms to make sure that the additional resources on different machines are fully utilized by clients.

Nevertheless, a system that runs on multiple machines does provide some benefits over one running a single large machine:

- *Increased redundancy.* If one machine fails, there are others that can take over the work. Machines might fail because of power or network outages, operating system crashes, application server failures, or even bugs in the application code itself.
- *Cost efficiency.* A network of smaller machines may have a better price/performance ratio than a single large machine has.

Many application products provide clustering services to enable the scaling out of applications. Again, though, clustering products vary considerably, and architects need to explore these differences carefully.

Distributed Transactions. Many EJB servers can coordinate transactions that involve multiple objects residing in various processes in a distributed system. Distributed transaction processing using the two-phase commit protocol is often essential in building enterprise-wide systems.

An architect designing an EJB system needs to consider carefully whether distributed transactions are necessary. This is because of the overhead involved in managing them, which increases as the number of transaction participants increases. If there is no need to coordinate the transaction across multiple resource managers (or databases), there is no need for the two-phase commit protocol.

Further, the transaction coordination and commit processes may involve several remote calls that pass over the network. These may be between the EJB server or container and an external transaction management process. If the distributed transaction implementation provided by the EJB server incurs additional remote calls in coordinating transactions, using distributed transactions can slow down an EJB system considerably, inhibiting overall system scalability.

Experience with various object technology management and J2EE implementations indicates large variations in distributed transaction management performance. This makes it important for application architects to fully understand the configuration and deployment options available with a given transaction service.

RESOURCE POOLING

Application resources, such as database connections and sockets, must be carefully managed in a distributed system. Resource pooling exploits the fact that not all clients need exclusive access to a resource at all times. With EJBs, not every bean needs a database connection for its exclusive use. It is much more efficient to configure a system so that database connections can be pooled and re-used for different client transactions.

When a database connection pool is used, the resulting connections required will be far less than the number of EJB components in a deployed system. Because database connections are expensive to create and manage, this architecture increases the overall application scalability. Furthermore, connections to the databases do not need to be reestablished continuously, thus improving application performance.

Resource pooling can be applied to other resources as well, such as socket connections and threads. Pooling of components simply means that a dedicated resource for each client is not necessary. Typical configurable parameters include container threads, session beans instances, entity bean cache size, and database connection pool size. All of these need to be configured appropriately to exhibit fast response times and high overall system throughput.

DEPENDENCE ON JAVA VIRTUAL MACHINE PERFORMANCE

In any Java application, the JVM is an important factor in performance tuning. Hence, to develop and deploy high-performing EJB server-side applications, several JVM configuration and performance tuning activities need to be considered.

JVM heap size is one important setting. The heap is a repository for Java objects and free memory. When the JVM runs out of memory in the heap, all execution in it ceases while a garbage collection algorithm goes through memory and frees space that is no longer required. This is an obvious performance hit because application code blocks during garbage collection. Thus, in an EJB application no server-side work can be done.

If heap size is huge, garbage collection will be infrequent; when it does kick in, however, it will take a much longer time, possibly long enough to disrupt normal system operations. Garbage collection can slow down (and sometime completely stop) server processing, giving the impression that the server is slow and unresponsive.

To appropriately set the JVM heap size, it is necessary to monitor the paging activities on the server machine. Paging is an expensive performance overhead and therefore should be avoided on application servers by increasing the JVM heap size to match the application's needs. Another way is to watch the garbage collector by using the `-gcverbose` compiler option. If incremental garbage collection is an option, it is almost always best to turn it on.

16.5 Summary

The creation of the J2EE multi-tier architecture was motivated by the business needs of Sun Microsystems. These business needs were influenced by the lessons of the CORBA model and by the competitive pressures of other proprietary distributed programming models, such as COM+ from Microsoft. J2EE features a server-side component framework for building enterprise-strength server-side Java applications, namely, Enterprise JavaBeans.

The J2EE/EJB specification is constantly expanding. Its ready-to-use services currently include transactions, security, naming, persistence, and resource management. These services enable the J2EE/EJB application programmer to focus on developing the business logic, thus removing the need to worry about low-level distribution details. J2EE/EJB achieves portability by using a common, portable language (Java) and by having precise contracts between components. It achieves performance and performance scalability via a number of mechanisms, including distributing applications across many processors (horizontal scaling), stateless session beans, and resource pools.

Despite the seeming simplicity of the J2EE/EJB programming model, there are many application-level architectural decisions that need to be carefully made. The various architectural tradeoffs must be analyzed and compared to derive an optimal design with respect to application quality requirements.

16.6 For Further Reading

There is an abundance of information about the J2EE/EJB architecture and specification. This includes Sun Microsystems's home page (*http://java.sun.com/j2ee*), which offers easy-to-follow tutorial material on J2EE, various white papers, and the J2EE/EJB specification itself. There are also numerous active forums focusing on the J2EE architecture and technology space, including one sponsored by The Middleware Company (*http://www.theserverside.com*).

16.7 Discussion Questions

1. An addition to the EJB component model version 2.0 is "message-driven beans." These are enterprise beans that allow J2EE applications to process messages asynchronously. What are some of the uses of such a component? What sort of new enterprise architecture possibilities do message-driven beans open up?

2. The J2EE/EJB specification uses many techniques that are actually just implementations of the "use an intermediary" tactic. Find as many distinct realizations of these instances as you can.

3. Consider the CelsiusTech case study presented in Chapter 15. Would J2EE/EJB be a good infrastructure choice for implementing this system? Justify your answer.