Distributed Objects and Components

Introduction

This essay will identify the differences between objects and components and what it means for a component to be distributed. It will also examine the Java 2 Enterprise Edition architecture and associated Enterprise Java Bean technologies.

Objects

A number of definitions of objects have been suggested in recent years. Booch defines an object as: "*an object has state, behaviour, and identity; the structure and behaviour of similar objects are defined in their common class*", in addition to this Rumbaugh adds that an object is "*a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand*". [1] The features often associated with objects are summarised below.

- An object has:
  - State
  - Operations
  - Attributes
- Provides an abstraction
- Represents something real
- Is self-contained
- Is clearly defined
- Helps to solve a problem

Components

A component can be thought of as a collection of objects that provide a set of services to other systems. They have many features in common with objects, but are less fine-grained. Examples of components include code that provides graphing facilities; provide network communication services; or perhaps a search and browsing service around a set of tables in a database.

Stal defines a component as "*A self-contained entity that exports functionality to its environment and may also import functionality from its environment using well-defined and open interfaces*". This definition highlights a number of differences from objects:

- Components are self-contained
- Provide services to other systems
- Use stated interfaces to interact with other systems

Components may be run either locally or in a distributed fashion. Many examples of locally run components exist and are commonly used to provide specific functionality to external programs. The Object Linking and Embedding (OLE) architecture was one of the first component frameworks in common use, and this enabled the use of specialised functionality from one program to be used in another program. An example of this is the use of Microsoft Excel spreadsheets in Microsoft Word.[2]

---

[1] [CETUS] Objects and Components (online reference)
[2] [CETUS] Objects and Components (online reference)

Distributed Components

Whereas objects and locally distributed components are located on the computer utilising them, distributed components are typically located on machines other than those making calls to them. [3]

Advantages of Distributed Components

The distribution of components has a number of advantages:

Load sharing

The use of distributed components facilitates interaction across machine boundaries, and this allows the system to harness the resources of multiple computers. Consequently, a number of machines may be involved in processing requests, potentially eliminating a performance bottleneck.

Increased availability

Distribution of components may result in increased system availability, with multiple instances of each object residing on machines in the network. If the instances are sharing load then the loss of one host will result in the load being distributed through the remaining machines. Replicated hosts may also be configured in a hot-swap or cold-swap arrangement. Each of these has different advantages and disadvantages.

Heterogeneity

Different hardware and software platforms store data in different ways. The most common difference is that some use a big-endian representation and others a little-endian representation. Distributed systems over come this problem by utilising a middleware layer that resolves the heterogeneity issues. The middleware enables distributed components to exist on multiple different platforms, and to still interact with one another.

Code reuse

It is intended that distributed components be constructed to supports reuse. This means that the components should be carefully designed so that they are useable in different contexts and systems. In practise this can be hard to achieve.

Disadvantages

In addition to these advantages, there are also some disadvantages with distributed components:

Multiple points of failure

By distributing the components of a system across multiple machines, the susceptibility of the system to failure can be increased. If all components of a distributed system are required to be available at any time network, machine failure can reduce the reliability of the system. With careful design these weaknesses can be reduced.

Complexity

Building a distributed system is a complex task. It involves the construction of a number of interacting elements and often these communicate asynchronously – making exhaustive testing almost impossible. The size of most distributed component systems means that multiple developers are involved in their construction, again this increases the complexity of the process, as the

[3] [EMM2000b] W. Emmerich, Engineering Distributed Objects

developers must keep up to date with all changes to interfaces and behaviours in the system.

Middleware

Middleware is the layer of software that mediates between an application and the network. It is responsible for managing the interaction between components distributed over heterogeneous computing platforms.

Types of Middleware

There are four categories of middleware technologies identified by Wolfgang Emmerich. They each exhibit different properties, and these classifications are examined below. [4]

Transaction-Oriented

Transactional middleware enables distributed components to ensure that an atomic operation either occurs completely or not at all. This is achieved by the use of a two-phase commit protocol. A transactional middleware does create some overhead, so if transactions are not required for an operation this is an unnecessary additional cost. Transaction-oriented middleware is commonly used in distributed databases.

Message-Oriented

This form of middleware provides message transmission, receipt and queuing services to distributed components. Messages are used to transmit service requests, updates and responses to requests. Messages are held in queues until they are de-queued by the receiving component and this ensures messages are not lost when a host is busy or unavailable.

Message-oriented middleware provides asynchronous communication between components - reducing the coupling between components. This leads to systems that are more scalable, with less tightly coupled components.

Procedural

The most common form of procedure-based middleware is Sun Microsystems' Remote Procedure Call middleware technology. This enables components on one computer to invoke a procedure or method call on a component located on another computer. The invocation and any associated parameters are marshalled into messages and these are sent from one machine to the other. The invocation semantics with a procedural middleware are commonly synchronous – with the invoker being blocked until a response is received.

Object-Oriented

Object-oriented middleware is based on the object-oriented programming paradigm, and extends the functionality of Procedural middleware to provide the additional facilities required for use in an object-oriented environment.

There are a number of different object-oriented middleware technologies widely used in industry. Two of these are each examined briefly below, and a detailed study of the Java 2 Enterprise Edition technology follows.

---

[4] [EMM2000] W. Emmerich, Software Engineering and Middleware: A Roadmap

CORBA

The Common Object Request Broker Architecture (CORBA) was defined by the Object Management Group and was intended to provide a standardized platform for which to construct distributed components. A CORBA application consists of a number of objects that store data and export functionality to other objects. There may be one or more instances of a particular object within the application, and each object type has an interface defined with the OMG Interface Definition Language (IDL). Whenever an invocation is performed, the appropriate method call from the object's interface is called and the arguments for the method are marshalled.

CORBA is language independent and has bindings to Java, C, C++ and many other common programming languages.[5]

DCOM

Microsoft developed the Component Object Model (COM) to support locally situated component, and this was later extended to support distributed invocations – resulting in the Distributed Component Object Model (DCOM). Microsoft implemented DCOM only for the Windows platform, implementations for other platforms have been produced independently, but have not proven popular. Consequently, DCOM is only commonly utilised within the Windows environment. There are several language bindings for DCOM – Microsoft Visual C++, Visual J++ and Visual Basic amongst others. As with CORBA, an IDL (in this case Microsoft IDL) is used to define the exported interfaces.

J2EE and EJB

The Java 2 Enterprise Edition provides a multiplatform distributed component platform. It consists of a set of services and protocols that are used at each level of a multi-tier distributed system. It provides services for all tiers (see Figure 1 below) of the J2EE distributed component architecture.

J2EE n-tier architecture

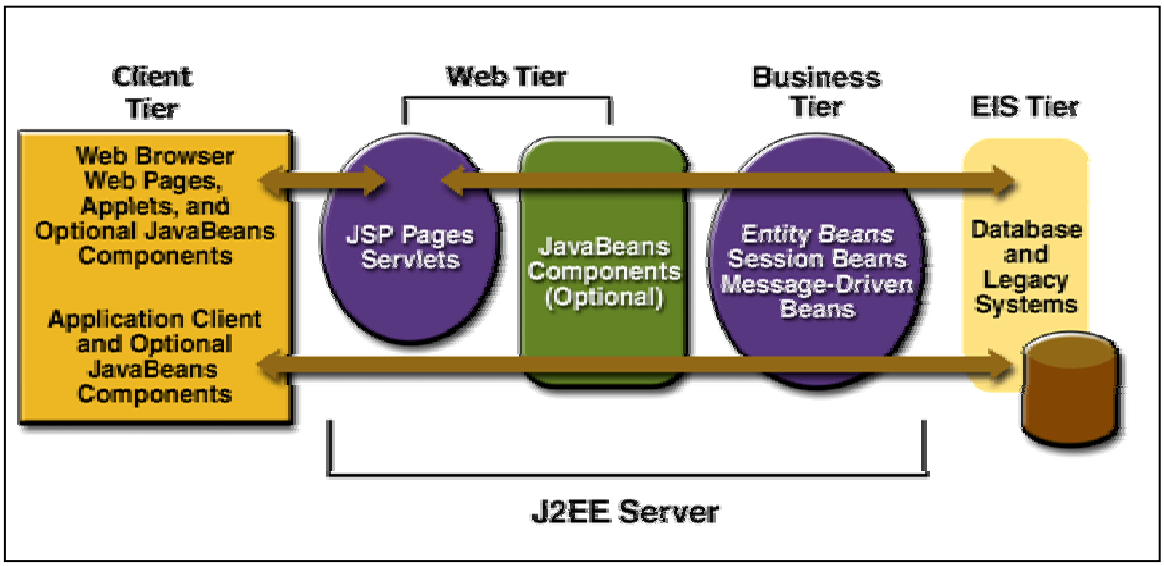

Figure 1 - The J2EE n-tier architecture[6]

[5] [OMG] Object Management Group; CORBA Basics
[6] taken from [SUN] Sun Microsystems; J2EE Tutorial

The client-tier provides an interface for the user to interact with the system. This may be in one of several forms: it could be a web browser (such as Netscape Navigator or Internet Explorer), a Java applet or Java-based client program.

The web-tier consists of Java Server Pages or servlets, which process and respond to requests from the client tier. The components of the web-tier obtain data and process information using the business-tier.

The business-tier is implemented in Enterprise JavaBeans (EJBs) and is commonly referred to as the Business Logic of the application. The EJBs are executed in a bean container – an application that controls the execution of JavaBeans and provides services such as transaction management, database connection pooling, security and authorisation facilities, remote machine connectivity, component persistence and replication. Further detail on the types of beans and bean containers is given later.

Behind the business-tier is the EIS-tier, which is comprised of the Enterprise Information Systems (EIS). This category contains; databases, transaction processing systems, resource-planning systems and other large-scale information systems and these will often be accessed by many different n-tier systems. For example, a bank may have one J2EE application for its traders to use, a system for external clients to trade, and another for reporting and monitoring functions, all utilising the same transaction processing system.

Enterprise JavaBeans

Enterprise JavaBeans (EJBs) are components that are used to provide the *business logic* for a J2EE application. They consist of a number of Java classes that have been packaged appropriately and are deployed to an EJB container that creates and manages the components. EJBs communicate using either Java Remote Method Invocation, or with Java Message Queues. There are a number of different types of EJBs, and these are examined below.

Session beans

A session bean is used to represent the state of a single interactive communication session between a client and the business-tier of the server. Session beans are transient; when a session is completed the associated session bean is discarded. If an application server fails, any session beans currently available to it are lost, as they are not stored in stable storage. There are two categories of session beans:

Stateful session beans hold the conversational state and one of these is required for each of the sessions that are currently open.

Stateless session beans hold no state (outside of calls) and receive all of their required input from the client-tier. These beans may be pooled and reused, thereby reducing the overheads of many clients accessing one server.

Entity beans

Entity beans provide an in-memory copy of long-term data. They are persistent, and are saved to stable storage to ensure they are preserved across machine crashes. Many clients may access an individual entity bean, and can find them by searching for the desired bean with the appropriate primary key.

An example of an entity bean is one that represents the historic prices of a stock. The data could be loaded from a database, and this entity bean could then be cached in memory and referenced by other entity bean.

Message-driven beans

Message beans were added to the Enterprise JavaBeans architecture later than session and entity beans. Originally, EJBs communicated using Java Remote Method Invocation; however, with the advent of the Java Message Service (JMS), message-driven beans were introduced. This form of bean is an asynchronous JMS message consumer, and to avoid tying up servers it uses a non-blocking primitive.

Bean Containers

As briefly mentioned above, EJB containers are responsible for the management of Enterprise JavaBeans. Before execution, an EJB component must be assembled into a J2EE application and deployed into its container. Each J2EE component has a configuration file associated with it (called a deployment descriptor), and this specifies the container settings for each of the EJBs and for the application as a whole.

The Bean container architecture is shown in Figure 2 below.

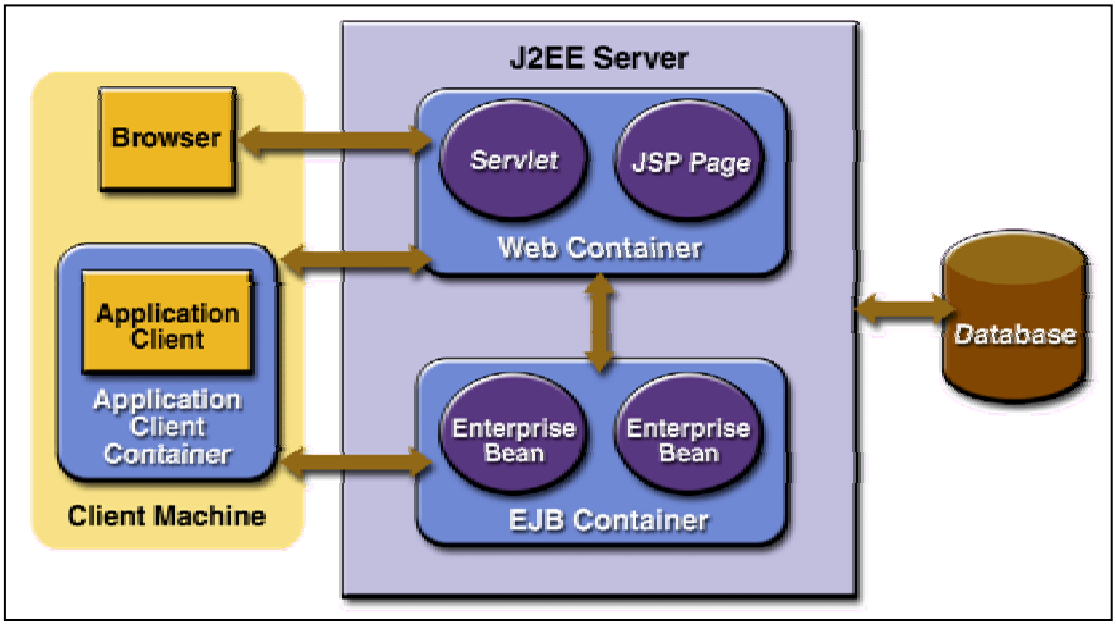

Figure 2 - EJB Containers[7]

Facilities provided

The container manages many aspects of the execution of each application. The configuration file enables components to customise the support services provided by the J2EE application server. These include transaction management, database connection management, security, remote host connectivity and EJB persistence.[8]

Advantages

The use of application servers such as EJB containers greatly simplifies the process of development and deployment of distributed components. It enables

[7] taken from [SUN] Sun Microsystems; J2EE Tutorial
[8] [SUN] Sun Microsystems; J2EE Tutorial

developers to concentrate on the functional and business-oriented aspects of the components they are developing, rather than having to consider concurrency controls, transactional behaviours, persistence, database connectivity and other complex issues. [9]

The EJB architecture extends the concept of reusability and takes it to a practical level, where whole components may be reused, rather than the piecemeal reuse of individual classes that occurs in class based object-oriented development.

Summary

This essay has identified many advantages of component-based development, with particular focus on a distributed environment. An overview of the CORBA and DCOM middleware architectures has been presented and the structure of the J2EE architecture, in particular Enterprise JavaBeans, has been examined in more detail. The importance of Bean Containers, and the range of facilities they provided, has also been looked at.

References

[EMM2000] Wolfgang Emmerich; Software Engineering and Middleware: A Roadmap; ACM Special Interest Group on Software Engineering, 2000

[EMM2000b] Wolfgang Emmerich; Engineering Distributed Objects; Wiley & Sons, 2000

[OMG] Object Management Group; CORBA Basics http://www.omg.org/gettingstarted/corbafaq.htm

[SUN] The J2EE Tutorial http://java.sun.com/j2ee/tutorial/

[CETUS] Objects and Components http://www.cetus-links.org

[9] [SUN] Sun Microsystems; J2EE Tutorial