3

Objects, Components, and the Web

This is the second chapter in our historical survey of middleware technology.

All the technologies described in Chapter 2 have their roots in the 1980s. At the end of that decade, however, there was a resurgence of interest in object-oriented concepts, in particular object-oriented (OO) programming languages. This led to the development of a new kind of OO middleware, one in which the requestor calls a remote object. In other words, it does something like an RPC call on an object method and the object may exist in another machine. It should be pointed out at once that of the three kinds of middleware discussed in Chapter 2—RPC/transactional, message queuing, and remote database access—OO middleware is a replacement for only the first of these. (The interest in OO has continued unabated since the first edition of this book, leading to a wide understanding of OO concepts. We therefore do not feel it necessary to describe the basic ideas.)

A notable example of OO middleware is the Common Object Request Broker Architecture (CORBA). CORBA is a standard, not a product, and was developed by the Object Management Group (OMG), which is a consortium of almost all the important software vendors and some large users. In spite of its provenance, it is one of those standards (the ISO seven-layered model is another) that has been influential in the computer industry and in academia, but is seldom seen in implementations. (A possible exception to this is the lower-level network protocol Internet Inter-ORB Protocol (IIOP), which has been used in various embedded network devices.) One reason for the lack of CORBA implementation was its complexity. In addition, interoperability among vendor CORBA implementations and portability of applications from one implementation to another were never very good. But possibly the major reason that CORBA never took off was the rise of component technology.

The key characteristics of a component are:

- It is a code file that can be either executed or interpreted.
- The run-time code has its own private data and provides an interface.
- It can be deployed many times and on many different machines.

In short, a component can be taken from one context and reused in another; one component can be in use in many different places. A component does not have to have an OO interface, but the component technology we describe in this book does. When executed or interpreted, an OO component creates one or more objects and then makes the interface of some or all of these objects available to the world outside the component.

One of the important component technologies of the 1990s was the Component Object Model (COM) from Microsoft. By the end of the 1990s huge amounts of the Microsoft software were implemented as COM components. COM components can be written in many languages (notably C++ and Visual Basic) and are run by the Windows operating system. Programs that wish to call a COM object don't have to know the file name of the relevant code file but can look it up in the operating system's registry. A middleware known as Distributed COM (DCOM) provides a mechanism to call COM objects in another Windows-operated machine across a network.

In the second half of the 1990s, another change was the emergence of Java as an important language. Java also has a component model, and its components are called JavaBeans. Instead of being deployed directly by the operating system, Java beans are deployed in a Java Virtual Machine (JVM), which runs the Java byte code. The JVM provides a complete environment for the application, which has the important benefit that any Java byte code that runs in one JVM will almost certainly run in another JVM. A middleware known as Remote Method Invocation (RMI) provides a mechanism to call Java objects in another JVM across a network.

Thus, the battle lines were drawn between Microsoft and the Java camp, and the battle continues today.

The first section in this chapter discusses the differences between using an object interface and using a procedure interface. Using object interfaces, in any technology, turns out to be surprisingly subtle and difficult. One reaction to the problems was the introduction of *transactional component middleware.* This term, coined in the first edition of this book, describes software that provides a container for components; the container has facilities for managing transactions, pooling resources, and other run-time functions to simplify the implementation of online transaction-processing applications. The first transactional component middleware was Microsoft Transaction Server, which evolved into COM+. The Java camp struck back with Enterprise JavaBeans (EJB). A more detailed discussion of transactional component middleware is in the second section.

One issue with all OO middleware is the management of sessions. Web applications changed the ground rules for sessions, and the final section of this chapter discusses this topic.

3.1 Using object middleware

Object middleware is built on the simple concept of calling an operation in an object that resides in another system. Instead of client and server, there are client and object.

To access an object in another machine, a program must have a reference pointing at the object. Programmers are used to writing code that accesses objects through pointers, where the pointer holds the memory address of the object. A reference is syntactically the same as a pointer; calling a local object through a pointer and calling a remote object through a reference are made to look identical. The complexities of using references instead of pointers and sending messages over the network are hidden from the programmer by the middleware.

Unlike in earlier forms of middleware, calling an operation on a remote object requires two steps: getting a reference to an object and calling an operation on the object. Once you have got a reference you can call the object any number of times.

We will illustrate the difference between simple RPC calls and object-oriented calls with an example. Suppose you wanted to write code to debit an account. Using RPCs, you might write something like this (We've used a pseudo language rather than C++ or Java because we hope it will be clearer.):

```
Call Debit(012345678, 100) ;    // where 012345678 is the account
                                // number and 100 is the amount
```

In an object-oriented system you might write:

```
Call AccountSet.GetAccount(012345678)    // get a reference to
    return AccountRef;                   // the account object
Call AccountRef.Debit(100);              // call debit
```

Here we are using an AccountSet object to get a reference to a particular account. (AccountSet is an object that represents the collection of all accounts.) We then call the debit operation on that account. On the face of it this looks like more work, but in practice there usually isn't much to it. What the client is more likely to do is:

```
Call AccountSet.GetAccount(X) return AccountRef;
Call AccountRef.GetNameAndBalance(....);
...display information to user
...get action to call – if it's a debit action then
Call AccountRef.Debit(Amt);
```

In other words, you get an object reference and then call many operations on the object before giving up the reference.

What this code segment does not explain is how we get a reference to the AccountSet object in the first place. In DCOM you might do this when you first connect to the component. In CORBA you may use a naming service that will take a name and look up an object reference for you. The subtleties in using objects across a network are discussed in more detail in the box entitled "Patterns for OO middleware."

Patterns for OO middleware

All middleware has an interface, and to use most middleware you must do two things: link to a resource (i.e., a service, a queue, a database), and call it by either passing it messages or call functions. OO middleware has the extra complexity of having to acquire a reference to an object before you can do anything. Three questions come to mind:

1. How do you get an object reference?
2. When are objects created and deleted?
3. Is it a good idea for more than one client to share one object?

In general, there are three ways to get an object reference:

1. A special object reference is returned to the client when it first attaches to the middleware. This technique is used by both COM and CORBA. The CORBA object returned is a system object, which you then interrogate to find additional services, and the COM object is an object provided by the COM application.
2. The client calls a special "naming" service that takes a name provided by the client and looks it up in a directory. The directory returns the location of an object, and the naming service converts this to a reference to that object. CORBA has a naming service (which has its own object interface). COM has facilities for interrogating the register to find the COM component but no standard naming service within the component.
3. An operation on one object returns a reference to another object. This is what the operation `GetAccount` in `AccountSet` did.

Broadly, the first two ways are about getting the first object to start the dialogue and the last mechanism is used within the dialogue.

Most server objects fall into one of the following categories:

- Proxy objects
- Agent objects
- Entrypoint objects
- Call-back objects

As an aside, there is a growing literature on what are called patterns, which seeks to describe common solutions to common problems. In a sense

what we are describing here are somewhat like patterns, but our aims are more modest. We are concentrating only on the structural role of distributed objects, not on how several objects can be assembled into a solution.

A proxy object stands in for something else. The AccountRef object is an example since it stands in for the account object in the database and associated account processing. EJB entity beans implement proxy objects. Another example is objects that are there on behalf of a hardware resource such as a printer. Proxy objects are shared by different clients, or at least look as if they are shared to the client.

A proxy object can be a constructed thing, meaning that it is pretending that such and such object exists, but in fact the object is derived from other information. For instance, the account information can be dispersed over several database tables but the proxy object might gather all the information in one place. Another example might be a printer proxy object. The client thinks it's a printer but actually it is just an interface to an e-mail system.

Agent objects are there to make the client's life easier by providing an agent on the server that acts on the client's behalf. Agent objects aren't shared; when the client requests an agent object, the server creates a new object. An important subcategory of agent objects is iterator objects. Iterators are used to navigate around a database. An iterator represents a current position in a table or list, such as the output from a database query, and the iterator supports operations like MoveFirst (move to the first row in the output set) and MoveNext (move to the next output row). Similarly, iterator objects are required for serial files access. In fact, iterators or something similar are required for most large-scale data structures to avoid passing all the data over the network when you need only a small portion of it. Other examples of agent objects are objects that store security information and objects that hold temporary calculated results.

An Entrypoint Object is an object for finding other objects. In the example earlier, the AccountSet object could be an entrypoint object. (As an aside, in pattern terminology an entrypoint object is almost always a creational pattern, although it could be a façade.)

A special case of an entrypoint object is known as a singleton. You use them when you want OO middleware to look like RPC middleware. The server provides one singleton object used by all comers. Singleton objects are used if the object has no data.

Call-back objects implement a reverse interface, an interface from server to client. The purpose is for the server to send the client unsolicited data. Call-back mechanisms are widely used in COM. For instance, GUI Buttons, Lists, and Text input fields are all types of controls in Windows and controls fire events. Events are implemented by COM call-back objects.

Some objects (e.g., entrypoint objects and possibly proxy objects) are never deleted. In the case of proxy objects, if the number of things you want proxies for is very large (such as account records in the earlier example), you may want to create them on demand and delete them when no longer

(*continued*)

Patterns for OO middleware (*cont.*)

needed. A more sophisticated solution is to pool the unused objects. A problem for any object middleware is how to know when the client does not want to use the object. COM provides a reference counter mechanism so that objects can be automatically deleted when the counter returns to zero. This system generally works well, although it is possible to have circular linkages. Java has its garbage-collection mechanism that searches through the references looking for unreferenced objects. This solves the problem of circular linkage (since the garbage collector deletes groups of objects that reference themselves but no other objects), but at the cost of running the garbage collector. These mechanisms have to be extended to work across the network with the complication that the client can suddenly go offline and the network might be disconnected.

From an interface point of view, object interfaces are similar to RPCs. In CORBA and COM, the operations are declared in an Interface Definition Language (IDL) file, as illustrated in Figure 3-1.

Like RPCs, the IDL generates a stub that converts operation calls into messages (this is marshalling again) and a skeleton that converts messages into operation calls. It's not quite like RPCs since each message must contain an object reference and may return an object reference. There needs to be a way of converting an object reference into a binary string, and this is different with every object middleware.

Unlike existing RPC middleware, the operations may also be called through an interpretive interface such as a macro language. There is no reason that RPCs shouldn't implement this feature; they just haven't. An interpretive interface requires some way of finding out about the operations at runtime and a way of building the parameter list. In CORBA, for instance, the information about an interface is stored in the interface repository (which looks like another object to the client program).

In object middleware, the concept of an interface is more explicit than in object-oriented languages like C++. Interfaces give enormous flexibility and strong encapsulation. With interfaces you really don't know the implementation because an interface is not the same as a class. One interface can be used in many classes. One interface can be implemented by many different programs. One object can support many interfaces.

In Java, the concept of an interface is made more explicit in the language, so it isn't necessary to have a separate IDL file.

So why would you think of using object middleware instead of, say, RPCs? There are two main reasons.

The first is simply that object middleware fits naturally with object-oriented languages. If you are writing a server in C++ or Visual Basic, almost all your data and logic will (or at least should) be in objects. If you are writing your server in

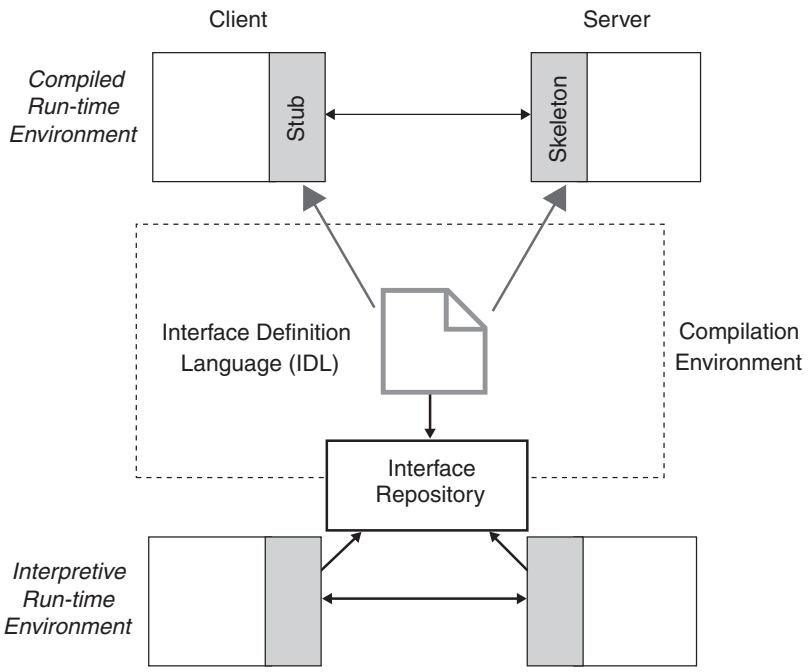

Figure 3-1 Object middleware compilation and interpretation

Java, all your data and code must be in objects. To design good object-oriented programs you start by identifying your objects and then you figure out how they interact. Many good programmers now always think in objects. Exposing an object interface through middleware is more natural and simpler to them than exposing a nonobject interface.

The second reason is that object middleware is more flexible. The fact that the interface is delinked from the server program is a great tool for simplification. For instance, suppose there is a single interface for security checking. Any number of servers can use exactly the same interface even though the underlying implementation is completely different. If there is a change to the interface, this can be handled in an incremental fashion by adding an interface to an object rather than by changing the existing interface. Having both the old and new interfaces concurrently allows the clients to be moved gradually rather than all at once.

3.2 Transactional component middleware

Transactional component middleware (TCM) covers two technologies: Microsoft Transaction Server (MTS), which became part of COM+ and is now incorporated in .NET, from Microsoft; and Enterprise JavaBeans (EJB) from the anti-Microsoft camp. OMG did release a CORBA-based standard for transactional component middleware, which was meant to be compatible with EJB, but extended the ideas into other languages. We will not describe this standard further since it has not attracted any significant market interest.

Transactional component middleware (TCM) is our term. TCM is about taking components and running them in a distributed transaction processing environment. (We discuss distributed transaction processing and transaction monitors in Chapter 2.) Other terms have been used, such as *COMWare* and *Object Transaction Manager (OTM).* We don't like *COMWare* because components could be used in a nontransactional environment in a manner that is very different from a transactional form of use, so having something about transactions in the title is important. We don't like *OTM* because components are too important and distinctive not to be included in the name; they are not the same as objects.

Transactional Component Middleware fits the same niche in object middleware systems that transaction monitors fit in traditional systems. It is there to make transaction processing systems easier to implement and more scalable.

The magic that does this is known as a container. The container provides many useful features, the most notable of which are transaction support and resource pooling. The general idea is that standard facilities can be implemented by the container rather than by forcing the component implementer to write lots of ugly system calls.

One of the advantages of Transactional Component Middleware is that the components can be deployed with different settings to behave in different ways. Changing the security environment is a case in point, where it is clearly beneficial to be able to change the configuration at deployment time. But there is some information that must be passed from developer to deployer, in particular the transactional requirements. For instance, in COM+ the developer must define that the component supports one of four transactional environments, namely:

1. Requires a transaction: Either the client is in transaction state (i.e., within the scope of a transaction) or COM+ will start a new transaction when the component's object is created.
2. Requires a new transaction: COM+ will always start a new transaction when the component's object is created, even if the caller is in transaction state.
3. Supports transactions: The client may or may not be in transaction state; the component's object does not care.
4. Does not support transactions: The object will not run in transaction state, even if the client is in transaction state.

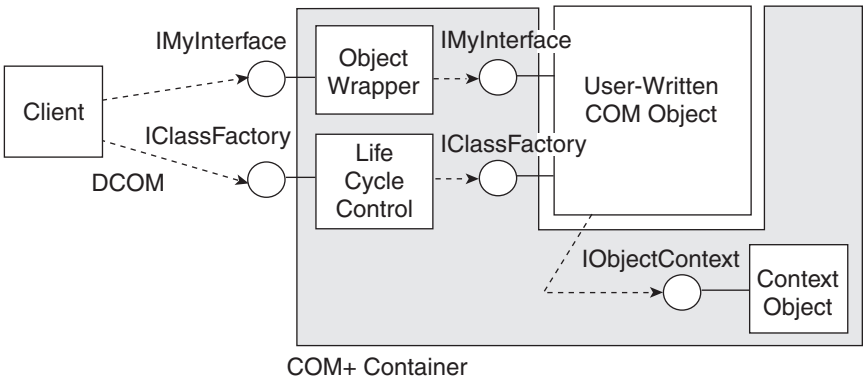

Figure 3-2 Transactional components in Microsoft COM+

In general, the first and third of these are commonly used. Note that the client can be an external program (perhaps on another system) or another component working within COM+. EJB has a similar set of features. Because the container delineates the transaction start and end points, the program code needs to do no more than commit or abort the transaction.

Figure 3-2 illustrates Microsoft COM+ and Figure 3-3 illustrates Enterprise JavaBeans. As you can see, they have a similar structure.

When a component is placed in a container (i.e., moved to a file directory where the container can access it and registered with the container), the administrator provides additional information to tell the container how to run the

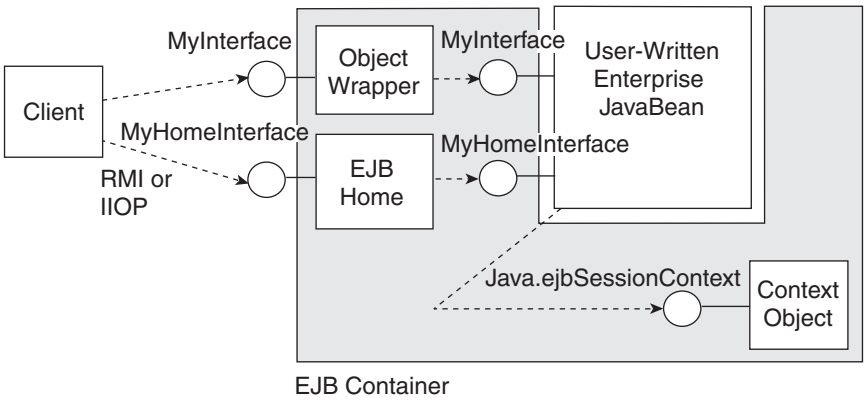

Figure 3-3 Transactional components in Enterprise JavaBeans

component. This additional information tells the system about the component's transactional and security requirements. How the information is provided depends on the product. In Microsoft COM+, it is provided by a graphical user interface (GUI), the COM+ Explorer. In the EJB standard, the information is supplied in eXtensible Markup Language (XML). For more information about XML, see the box about XML in Chapter 4.

A client uses the server by calling an operation in the IClassFactory (COM+) or MyHomeInterface (EJB) interface to create a new object. The object's interface is then used directly, just as if it were a local object. In Figures 3-2 and 3-3 you see that the client reference does not point at the user written component but at an object wrapper. The structure provided by the container provides a barrier between the client and the component. One use of this barrier is security checking. Because every operation call is intercepted, it is possible to define security to a low level of granularity.

The other reason for the object wrapper is performance. The object wrapper makes it possible to deactivate the component objects without the client's knowledge. The next time the client tries to use an object, the wrapper activates the object again, behind the client's back, so to speak. The purpose of this is to save resources. Suppose there are thousands of clients, as you would expect if the application supports thousands of end users. Without the ability to deactivate objects, there would be thousands of objects, probably many thousands of objects because objects invoke other objects. Each object takes memory, so deactivating unused objects makes an enormous difference to memory utilization.

Given that objects come and go with great rapidity, all the savings from the efficient utilization of memory would be lost by creating and breaking database connections, because building and breaking down database connections is a heavy user of system resources. The solution is connection pooling. There is a pool of database connections, and when the object is deactivated the connection is returned to the pool. When a new object is activated, it reuses an inactive connection from the pool. Connection pooling is also managed by the container.

The next obvious question is, when are objects deactivated? Simply deleting objects at any time (i.e., when the resources are a bit stretched) could be dangerous because the client might be relying on the component to store some information. This is where COM+ and EJB differ.

3.2.1 COM+

In COM+, you can declare that the object can be deactivated after every operation or at the end of a transaction. Deactivation in COM+ means elimination; the next time the client uses the object, it is recreated from scratch.

Deactivating after every operation brings the system back to the level of a traditional transaction monitor, because at the beginning of every operation the code will find that all the data attributes in the object are reset to their initial state.

Deactivating at the end of every transaction allows the client to make several calls to the same object, for instance, searching for a record in the database in one call and updating the database in another call. After the transaction has finished, the object is deactivated.

A traditional feature of transaction monitors is the ability to store data on a session basis, and you may have noticed that there is no equivalent feature in COM+. Most transaction monitors have a data area where the transaction code can stash data. The next time the same terminal runs a transaction, the (possibly different) transaction code can read the stash. This feature is typically used for storing temporary data, like remembering the account number this user is working on. Its omission in COM+ has been a cause of much argument in the industry.

3.2.2 EJB

Enterprise JavaBeans is a standard, not a product. There are EJB implementations from BEA, IBM, Oracle, and others. The network connection to EJB is the Java-only Remote Method Invocation (RMI) and the CORBA interface IIOP. IIOP makes it possible to call an EJB server from a CORBA client.

EJB components come in two flavors, session beans and entity beans. Each has two subflavors. Session beans are logically private beans; that is, it is as if they are not shared across clients. (They correspond roughly to what we describe as agent objects in the previous box entitled "Patterns for OO middleware.") The two subflavors are:

- Stateless session beans: All object state is eliminated after every operation invocation.
- Stateful session beans: These hold state for their entire life.

Exactly when a stateful session bean is "passivated" (the EJB term for deactivated) is entirely up to the container. The container reads the object attributes and writes them to disk so that the object can be reconstituted fully when it is activated. The stateful bean implementer can add code, which is called by the passivate and activate operations. This might be needed to attach or release some external resource.

The EJB container must be cautious about when it passivates a bean because if a transaction aborts, the client will want the state to be like it was before the transaction started rather than what it came to look like in the middle of the aborted transaction. That in turn means that the object state must be saved during the transaction commit. In fact, to be really safe, the EJB container has to do a two-phase commit to synchronize the EJB commit with the database commit. (In theory it would be possible to implement the EJB container as part of the database software and manage the EJB save as part of the database commit.)

Entity beans were designed to be beans that represent rows in a database.

Normally the client does not explicitly create an entity bean but finds it by using a primary key data value. Entity beans can be shared.

The EJB specification allows implementers to cache the database data values in the entity bean to improve performance. If this is done, and it is done in many major implementations, it is possible for another application to update the database directly, behind the entity bean's back so to speak, leaving the entity bean cache holding out-of-date information. This would destroy transaction integrity. One answer is to allow updates only through the EJBs, but this is unlikely to be acceptable in any large-scale enterprise application. A better solution is for the entity bean not to do caching, but you must ensure that your EJB vendor supports this solution.

The two subflavors of entity beans are:

- Bean-managed persistence: The user writes the bean code.
- Container-managed persistence: The EJB automatically maps the database row to the entity bean.

Container-managed persistence can be viewed as a kind of 4GL since it saves a great deal of coding.

3.2.3 Final comments on TCM

When EJBs and COM+ first appeared, there was a massive amount of debate about which was the better solution. The controversy rumbles on. An example is the famous Pet Store benchmark, the results of which were published in 2002. The benchmark compared functionally identical applications implemented in J2EE (two different application servers) and .NET. The results suggested that .NET performed better and required fewer resources to develop the application. This unleashed a storm of discussion and cries of "foul!" from the J2EE supporters.

In our opinion, the controversy is a waste of time, for a number of reasons. A lot of it arose for nontechnical reasons. The advocates—*disciples* might be a better word—of each technology would not hear of anything good about the other or bad about their own. The debate took on the flavor of a theological discussion, with the protagonists showing all the fervor and certainty of Savonarola or Calvin. This is ultimately destructive, wasting everyone's time and not promoting rational discussion. Today there are two standards, so we have to live with them. Neither is likely to go away for lack of interest, although the next great idea could replace both of them. And is it bad to have alternatives? Many factors contribute to a choice of technology for developing applications (e.g., functional requirements, performance, etc.). The two technologies we have been discussing are roughly equivalent, so either could be the right choice for an enterprise. The final decision then comes down to other factors, one of which is the skill level in the organization concerned. If you have a lot of Java expertise, EJB is the better choice. Similarly, if you have a lot of Microsoft expertise, choose COM+.

There are, of course, legitimate technical issues to consider. For example, if you really do want operating system independence, then EJB is the correct choice; the Microsoft technology works only with Windows. If you want language independence, you cannot choose EJBs because it supports Java only. There may also be technical issues about interworking with existing applications, for which a gateway of some form is required. It could be that one technology rather than the other has a better set of choices, although there are now many options for both.

Both technologies have, of course, matured since their introduction, removing some reasonable criticisms; the holes have been plugged, in other words. And a final point we would like to make is that it is possible to produce a good application, or a very bad one, in either of these technologies—or any other, for that matter. Producing an application with poor performance is not necessarily a result of a wrong choice of technology. In our opinion, bad design and implementation are likely to be much greater problems, reflecting a general lack of understanding both of the platform technologies concerned and the key requirements of large-scale systems. Addressing these issues is at the heart of this book.

Transaction component middleware is likely to remain a key technology for some time. COM+ has disappeared as a marketing name but the technology largely remains. It is now called Enterprise Services and is part of Microsoft .NET. More recent developments, which have come very much to the fore, are service orientation and service-oriented architectures in general, and Web services in particular, which we discuss in the next chapter.

3.3 Internet Applications

In the latter part of the 1990s, if the press wasn't talking about the Microsoft/Java wars, it was talking about the Internet. The Internet was a people's revolution and no vendor has been able to dominate the technology. Within IT, the Internet has changed many things, for instance:

- It hastened (or perhaps caused) the dominance of TCP/IP as a universal network standard.
- It led to the development of a large amount of free Internet software at the workstation.
- It inspired the concept of thin clients, where most of the application is centralized. Indeed, the Internet has led to a return to centralized computer applications.
- It led to a new fashion for data to be formatted as text (e.g., HTML and XML). The good thing about text is that it can be read easily and edited by a simple editor (such as Notepad). The bad thing is that it is wasteful of space and requires parsing by the recipient.

- It changed the way we think about security (discussed in Chapter 10).
- It liberated us from the notion that specific terminals are of a specific size.
- It led to a better realization of the power of directories, in particular Domain Name Servers (DNS) for translating Web names (i.e., URLs) into network (i.e., IP) addresses.
- It led to the rise of intranets—Internet technology used in-house—and extranets—private networks between organizations using Internet technology.
- It has to some extent made people realize that an effective solution to a problem does not have to be complicated.

Internet applications differ from traditional applications in at least five significant ways.

First, the user is in command. In the early days, computer input was by command strings and the user was in command. The user typed and the computer answered. Then organizations implemented menus and forms interfaces, where the computer application was in command. The menus guide the user by giving them restricted options. Menus and forms together ensure work is done only in one prescribed order. With the Internet, the user is back in command in the sense that he or she can use links, Back commands, Favorites, and explicit URL addresses to skip around from screen to screen and application to application. This makes a big difference in the way applications are structured and is largely the reason why putting a Web interface on an existing menu and forms application may not work well in practice.

Second, when writing a Web application you should be sensitive to the fact that not all users are equal. They don't all have high-resolution, 17-inch monitors attached to 100Mbit or faster Ethernet LANs. Screens are improving in quality but new portable devices will be smaller again. And in spite of the spread of broadband access to the Internet, there are, and will continue to be, slow telephone-quality lines still in use.

Third, you cannot rely on the network address to identify the user, except over a short period of time. On the Internet, the IP address is assigned by the Internet provider when someone logs on. Even on in-house LANs, many organizations use dynamic address allocation (the DHCP protocol), and every time a person connects to the network he or she is liable to get a different IP address.

Fourth, the Internet is a public medium and security is a major concern. Many organizations have built a security policy on the basis that (a) every user can be allocated a user code and password centrally (typically the user is given the opportunity to change the password) and (b) every network address is in a known location. Someone logging on with a particular user code at a particular location is given a set of access rights. The same user at a different location may not have the same access rights. We have already noted that point (b) does not hold on the Internet, at least not to the same precision. Point (a) is also suspect; it is much

more likely that user code security will come under sustained attack. (We discuss these points when we discuss security in Chapter 10.)

Fifth and finally, it makes much more sense on the Internet to load a chunk of data, do some local processing on it, and send the results back. This would be ideal for filling in big forms (e.g., a tax form). At the moment these kinds of applications are handled by many short interactions with the server, often with frustratingly slow responses. We discuss this more in Chapters 6 and 13.

Most nontrivial Web applications are implemented in a hardware configuration that looks something like Figure 3-4.

You can, of course, amalgamate the transaction and database server with the Web servers and cut out the network between them. However, most organizations don't do this, partly because of organizational issues (e.g., the Web server belongs to a different department). But there are good technical reasons for making the split, for instance:

- You can put a firewall between the Web server and the transaction and database server, thus giving an added level of protection to your enterprise data.
- It gives you more flexibility to choose different platforms and technology from the back-end servers.
- A Web server often needs to access many back-end servers, so there is no obvious combination of servers to bring together.

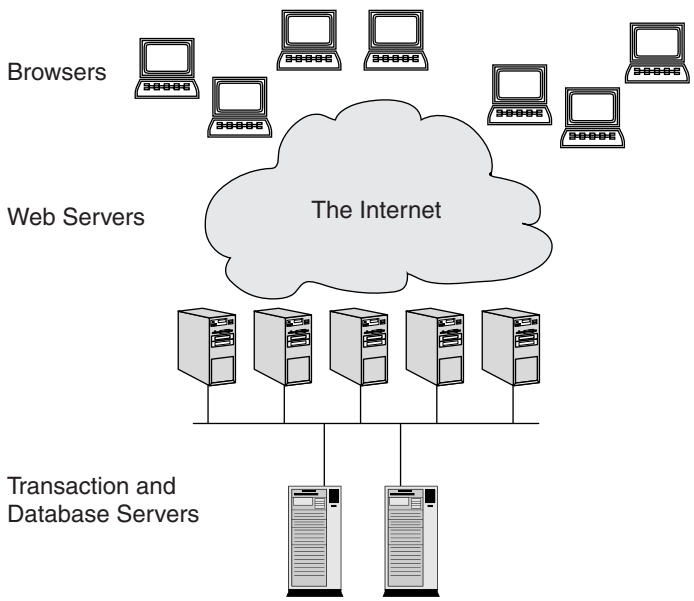

Figure 3-4 Web hardware configuration

Web servers are easily scalable by load balancing across multiple servers (as long as they don't hold session data). Others, for example, database servers, may be harder to load balance. By splitting them, we have the opportunity to use load balancing for one and not the other. (We discuss load balancing in Chapter 8.)

The Transactional Component Middleware was designed to be the middleware between front- and back-end servers.

Many applications require some kind of session concept to be workable. A session makes the user's life easier by

- Providing a logon at the start, so authentication need be done only once.
- Providing for traversal from screen to screen.
- Making it possible for the server to collect data over several screens before processing.
- Making it easier for the server to tailor the interface for a given user, that is, giving different users different functionality.

In old-style applications these were implemented by menu and forms code back in the server. Workstation GUI applications are also typically session-based; the session starts when the program starts and stops when it stops. But the Web is stateless, by which we mean that it has no built-in session concept. It does not remember any state (i.e., data) from one request to another. (Technically, each Web page is retrieved by a separate TCP/IP connection.) Sessions are so useful that there needs to be a way to simulate them. One way is to use applets. This essentially uses the Web as a way of downloading a GUI application. But there are problems.

If the client code is complex, the applet is large and it is time consuming to load it over a slow line. The applet opens a separate session over the network back to the server. If the application is at all complex, it will need additional middleware over this link.

A simple sockets connection has the specific problem that it can run foul of a firewall since firewalls may restrict traffic to specific TCP port numbers (such as for HTTP, SMTP, and FTP communication). The applet also has very restricted functionality on the browser (to prevent malicious applets mucking up the workstation).

Java applets have been successful in terminal emulation and other relatively straightforward work, but in general this approach is not greatly favored. It's easier to stick to standard HTML or dynamic HTML features where possible.

An alternative strategy is for the server to remember the client's IP address. This limits the session to the length of time that the browser is connected to the network since on any reconnect it might be assigned a different IP address. There is also a danger that a user could disconnect and another user could be assigned the first user's IP address, and therefore pick up their session!

A third strategy is for the server to hide a session identifier on the HTML page in such a way that it is returned when the user asks for the next screen

(e.g., put the session identifier as part of the text that is returned when the user hits a link). This works well, except that if the user terminates the browser for any reason, the session is broken.

Finally, session management can be done with cookies. Cookies are small amounts of data the server can send to the browser and request that it be loaded on the browser's disk. (You can look at any text in the cookies with a simple text editor such as Notepad.) When the browser sends a message to the same server, the cookie goes with it. The server can store enough information to resume the session (usually just a simple session number). The cookie may also contain a security token and a timeout date. Cookies are probably the most common mechanism for implementing Web sessions. Cookies can hang around for a long time; therefore, it is possible for the Web application to notice a single user returning again and again to the site. (If the Web page says "Welcome back <your name>", it's done with cookies.) Implemented badly, cookies can be a security risk, for instance, by holding important information in clear text, so some people disable them from the browser.

All implementations of Web sessions differ from traditional sessions in one crucial way. The Web application server cannot detect that the browser has stopped running on the user's workstation.

How session state is handled becomes an important issue. Let us take a specific example—Web shopping cart applications. The user browses around an online catalogue and selects items he wishes to purchase by pressing an icon in the shape of a shopping cart. The basic configuration is illustrated in Figure 3-4. We have:

- A browser on a Web site
- A Web server, possibly a Web server farm implemented using Microsoft ASP (Active Server Pages), Java JSP (JavaServer Pages), or other Web server products
- A back-end transaction server using .NET or EJB

Let us assume the session is implemented by using cookies. That means that when the shopping cart icon is pressed, the server reads the cookie to identify the user and displays the contents of the shopping cart. When an item is added to the shopping cart, the cookie is read again to identify the user so that the item is added to the right shopping cart. The basic problem becomes converting cookie data to the primary key of the user's shopping cart record in the database. Where do you do this? There are several options of which the most common are:

- Do it in the Web server.
- Hold the shopping cart information in a session bean.
- Put the user's primary key data in the cookie and pass it to the transaction server.

The Web server solution requires holding a lookup table in the Web server to convert cookie data value to a shopping cart primary key. The main problem is

that if you want to use a Web server farm for scalability or resiliency, the lookup table must be shared across all the Web servers. This is possible, but it is not simple. (The details are discussed Chapter 7.)

Holding the shopping cart information in a session bean also runs into difficulties when there is a Web server farm, but in this case the session bean cannot be shared. This is not an insurmountable problem because in EJB you can read a *handle* from the object and store it on disk, and then the other server can read the handle and get access to the object. But you would have to ensure the two Web servers don't access the same object at the same time. Probably the simplest way to do this is to convert the handle into an object reference every time the shopping cart icon is pressed. Note that a consequence of this approach is that with 1,000 concurrent users you would need 1,000 concurrent session beans. A problem with the Web is that you don't know when the real end user has gone away, so deleting a session requires detecting a period of time with no activity. A further problem is that if the server goes down, the session bean is lost.

The simplest solution is to store the shopping cart information in the database and put the primary key of the user's shopping cart directly in the cookie. The cookie data is then passed through to the transaction server. This way, both the Web server and the transaction server are stateless, all these complex recovery problems disappear, and the application is more scalable and efficient.

In our view, stateful session beans are most useful in a nontransactional application, such as querying a database. We can also envisage situations where it would be useful to keep state that had nothing to do with transaction recovery, for instance, for performance monitoring. But as a general principle, if you want to keep transactional state, put it in the database.

On the other hand, keeping state during a transaction is no problem as long as it is reinitialized if the transaction aborts, so the COM model is a good one. To do the same in EJB requires using a stateful session bean but explicitly reinitializing the bean at the start of every transaction.

But you needed session state for mainframe transaction monitors, why not now? Transaction monitors needed state because they were dealing with dumb terminals, which didn't have cookies—state was related to the terminal identity. Also, the applications were typically much more ruthless about removing session state if there was a recovery and forcing users to log on again. For instance, if the network died, the mainframe applications would be able to log off all the terminals and remove session state. This simplified recovery. In contrast, if the network dies somewhere between the Web server and the browser, there is a good chance the Web server won't even notice. Even if it does, the Web server can't remove the cookie. In the olden days, the session was between workstation and application; now it is between cookie and transaction server. Stateful session beans support a session between the Web server and the transaction server, which is only part of the path between cookie and transaction server. In this case, having part of an implementation just gets in the way.

Entity beans, on the other hand, have no such problems. They have been criticized for forcing the programmer to do too many primary key lookup operations on the database, but we doubt whether this performance hit is significant.

3.4 Summary

Key points to remember:

- Transaction Component Middleware (TCM) is the dominant technology today for transaction processing applications. The two leading TCMs are Enterprise JavaBeans (EJB) and .NET Enterprise Services (formerly COM+).
- These two dominant TCM technologies both use OO interfaces. OO interfaces have greater flexibility than older interface styles like RPC and fit well with OO programming languages. But there is a cost in greater complexity because there are objects to be referenced and managed.
- TCMs are preferable to older OO middleware styles like DCOM and CORBA because developing transactional applications is easier (there is much less to do) and the software provides object and database connection pooling, which improves performance.
- Web browsers are significantly different from older GUI applications or green-screen terminals. The browser user has more control over navigation, the server can make far fewer assumptions on the nature of the device, and session handling is different. In particular, the Web application server has no idea when the browser user has finished using the application.
- While there are many fancy features for session handling in EJBs, a simple approach using stateless sessions is usually best. The old adage, KISS—Keep it Simple, Stupid—applies.
- In large organizations, the chances are you will have to work with both .NET and Java for the foreseeable future.