Alternative Technologies

Application Servers: An Introduction for Database Experts

by David McGoveran

Over the past few years, many of my database colleagues have repeatedly expressed bewilderment when presented with the terminology and concepts common to today's J2EE (Java 2 Enterprise Edition) application servers. In this article, we'll introduce the features and functions that an application server provides by drawing analogies to the features and functions found in a DBMS. For our purposes, we're talking about WebSphere and DB2. Although I will limit discussion to WebSphere as a J2EE application server, it is important to understand that WebSphere is an IBM brand and, as such, covers a suite of products of which the J2EE application server is one. Hopefully, the importance of the WebSphere application server to DB2 will become apparent despite the glosses forced by space limitations.

Essential Background

Over the past decade, the character of enterprise applications has changed radically. We have seen a shift from applications developed with strongly coupled, functionally defined modules to those developed with more loosely coupled and reusable business objects. The rise of distributed, multi-platform componentization and the maturation of object oriented programming required the evolution of a deployment environment with more sophistication than traditional operating systems. The capabilities of early distributed application environments such as CORBA attempted to address many of the problems inherit in multi-tier application management, but the complexity of application development using traditional programming languages with CORBA – among other factors – limited its acceptance.

Having audited, designed, and developed many distributed applications over the last twenty or so years and taught many seminars on developing client/server applications between 1984 and 1996 and more recently on effective combinations of RDBMS and application servers, I can personally attest to the difficulty that developers have implementing scalable, high performance, highly available, and manageable distributed applications. The common (albeit incorrect) perception of the client/server approach as a physical two-tier architecture with the database server on the back end led to many problems. Developers seldom understood multi-threading, transaction management, synchronization, data integrity, or other issues, and often were unable to invoke the operating specific system services that would help them manage running applications. "Fat-client" applications often redundantly implemented functionality that should have been shared across applications. Furthermore, most application design and development tools favored a specific language or DBMS.

In the attempt to remove the limitations inherent in a physical, two-tier architecture, multi-tier architectures became more common. Initially, most multi-tier architectures simply separated three tiers – presentation, application or business logic, and database, an architecture that is at least conceptually familiar today. Multi-tier architectures introduce considerable development and runtime management complexity. With fixed numbers of systems in each tier, application design that uses those resources fairly efficiently, and scales and performs reasonably well within certain load bounds is feasible. If those numbers of systems can change so that capacity changes, either the application code must be altered or else those aspects of the application that affect capacity and load must be managed external to the application. Needless to say, few developers were capable of solving these problems. That's where a J2EE application server like WebSphere comes in: It helps both developers and administrators solve these problems.

Two primary environments are provided by a J2EE application server: design time (which includes both design and development) and runtime. The design time enforces standard software architecture guidelines, while the runtime provides the services that must be used to implement those features and functions that affect system wide performance and scalability. Together, these environments remove much of the aforementioned distributed application complexity from the concerns of the programmer.

The design time environment, or IDE (integrated development environment), promotes a particular architecture for the application code and targets specific runtime services and facilities. Some applications servers have attempted to support development in a variety of languages. The most successful application servers today (based on market share) are either J2EE-compliant (e.g., BEA WebLogic and IBM WebSphere) or proprietary (e.g., Microsoft .NET). There are many reasons for this, but in certain respects, Java is to application servers as SQL is to database servers. WebSphere Studio, WebSphere's design time environment, has facilities for developing Java applications that access DB2 and a number of other RDBMS products, as well as XML, Web services, asynchronous messaging, and process model driven development. Control flow between components and Web services can be defined graphically in a process model and then automatically generated.

I won't spend much more time explaining WebSphere Studio. However, in order to understand the runtime environment, you need to understand a couple of key features of Java that support object oriented componentization and distribution. You probably already know Java is an object oriented programming language and that it is mostly platform independent. You might also know that this platform independence is accomplished by implementing a platform-specific virtual machine (the JVM) that hides platform differences and creates a predictable Java runtime environment. Part of Java's value is that its very structure encourages an object oriented discipline that was hard to maintain in, for example, C++.

Java makes lots of programming tasks like exception handling, multithreading, and client/server programming (via remote method invocation) relatively easy. Standard packages are included for network I/O, user interfaces, SQL, and so on. Much in the same way that a DBA need not develop the code for various access methods, data structures, or relational operators, Java provides a variety of classes (such as hash tables) that a developer can reuse. Java also removes certain capabilities from the reach of the programmer, instead performing these functions automatically in a disciplined manner. For example, Java does not provide a programmer explicit control over memory allocation, deallocation, or garbage collection.

Although Java avoids many of the problems often seen in enterprise applications that were developed in C or COBOL, all this power still makes it easy to develop programs that do not behave well when it comes to enterprise application issues. The EJB (Enterprise Java Bean) distributed component architecture was created to solve this problem. In terms of code encapsulation and reuse, an EJB is similar to a database stored procedure: It is stored on and executed by the server. By conforming to the EJB specification, application behavior with respect to distributed implementation, transaction management, and security is predictable and much easier to make scalable. EJBs are part of J2EE. A J2EE application server IDE automatically generates code to interact with the system services provided application server, including the code that handles distributed instance registration with and invocation via the object request broker.

Java uses separate files to package each Java class. A Java application consists of a collection of classes (possibly following the EJB component specification), and possibly other resources. These are usually packaged together in compressed file called a JAR (Java Application Resource) file. WebSphere also generates WAR (web application resource) and EAR (enterprise application resource) files.

The J2EE application server runtime environment can be thought of as a layer on top of the operating system which provides portability via common APIs and a variety of management services for distributed applications. The specific capabilities of the runtime environment supported by an application server vary considerably in sophistication from product to product.

Application Servers By Analogy

Relational database management systems (RDBMSs), and versions of DB2 in particular, provide a great many services which we sometimes take for granted. Those of us who have worked as database administrators or who have worked on the internals of an RDBMS are almost certainly aware of those services and how to make appropriate use of the them. As we will see, these services have a functional parallel in those provided by J2EE application servers like WebSphere. Two caveats are in order before we begin. First, although DB2 and other RDBMS products provide direct support for Java and XML, we will ignore these capabilities with respect to forming analogies. Second, when I speak of an RDBMS, I will assume that it has distributed database functionality, and therefore that the database is distributed over multiple physical servers.

An SQL statement is the smallest unit of work an RDBMS processes on request from an external agent such as an application or user. Similar to Java application code, prior to execution, SQL statements must be parsed, interpreted, and optimized. In that process, the RDBMS will identify data element references in the SQL statement and determine how to resolve those references. In combination with access methods chosen, it will locate the relevant data elements so that they can be accessed at execution time. In much the same way, a unit of work to be executed by a J2EE application server must be parsed, interpreted, and optimized and any objects references in the Java code identified and resolved. The location of an object is determined by a facility within the Application Server (part of the infrastructure benefits mentioned above) called an *object request broker (ORB)*, which then forwards the request to the server on which the object is located. In much the same way, a distributed RDBMS uses the system catalog to determine the location of data and may break a SQL statement into sub-statements which it then forwards to the appropriate database server. So, like a distributed RDBMS, a J2EE application server supports a kind of location transparency, except with respect to application components (EJBs in this case) instead of data.

Distribution and Replication

Curiously, there is an important difference between how and when an RDBMS supports data replication and how and when a J2EE application server supports replication of EJB instances. I want to emphasize that this difference is a failing of current distributed RDBMS technology, and not one of principle, theory, or practicality. In principle, a distributed RDBMS should support replication transparency. This means that the RDBMS should support controlled replication of data in a way that is completely transparent to the application. It would then be able to make data copies and distribute them automatically so as to maximize performance. In practice, current technology does not provide replication transparency and the distribution of replicated data must be done manually (i.e., the RDBMS does not automatically determine the need for a replicate, create it, and synchronize the data). .

Like a distributed RDBMS, an application server can run on multiple physical servers. Usually, these servers form a cluster. An application server has some ability to control redundancy in an effort to manage performance. To this end, it can create new (and destroy old) instances of an object on the various physical servers it uses and determine which copy of the object to access at runtime. This somewhat complicates the task of an application server's object request broker over the job done by the RDBMS since it must also determine which copy of an object to use.

Load Balancing, Recovery, Failover, and Cache

When application servers are clustered for scalability and high availability, EJB invocation is managed automatically. Like a clustered RDBMS server, an application server uses various algorithms to balance request load across the physical servers in the cluster. Load can be balanced, for example, by a load factor algorithm or round-robin scheduling. If an application or transaction fails, recovery is handled by the application server. If an application server node in the cluster fails, failover can often be automatic

As with an RDBMS, the application developer does not have direct control of memory management. A J2EE application server thus manages cache for all applications. In the case of WebSphere, cache is managed across nodes in the cluster so that an update on one node is propagated to all nodes in the cluster.

Transaction Management

Unless you're familiar with CICS or another TP monitor, you probably take the transaction management capabilities of an RDBMS pretty much for granted, controlling commit and rollback explicitly to set transaction boundaries. The details of transaction management are pretty transparent to the application. An application server like WebSphere extends this capability to non-database operations much like a traditional TP monitor does, but with some important differences. In particular, transaction boundaries in an EJB application are controlled not by explicit commit and rollback commands (this is possible but denigrated), but by one of several conventions determined by externally settable attributes.

For example, when an EJB is invoked, its deployment attributes determine whether (a) it participates in a enveloping transaction, (b) does not participate in an enveloping transaction, (c) requires that it be invoked within a transaction, (d) automatically beings a new transaction, (e) must be invoked within a transaction, or (f) must not be within a transaction. These options are controlled in part by the type of EJB involved. *Container Managed Persistence* is commonly used for entity beans – EJBs that include database access – meaning that the code does not manage the bean's state but that the container automatically generates the necessary code to access a database or so-called persistence server. However, it is important to understand that the developer may optionally choose to use *Bean Managed Persistence*, which just means that the developer of the bean determines how persistence is to be managed. Conceptually, a container for a collection of EJBs is an intermediary between those EJBs and the application server, and provides a managed environment for those EJBs with the necessary transaction management, location transparency, and multi-threading. In practice, containers are provided implicitly with the application server and do not need to be developed. You can think of a container as a facility of the application server.

A data source like DB2 is bound to a Java Naming and Directory Interface (JNDI) name. JNDI is the API that provides uniform access to naming and directory services, similar to an API for managing an RDBMS system catalog. For EJBs that use container managed persistence, fields in the EJB are mapped to table columns, and this mapping is used by the container to move data between an instance of the EJB and the database with automatically generated SQL. For EJBs that use bean managed persistence, the developer both writes the SQL and codes the mapping directly. In WebSphere, binding, mapping, and connection configuration are done with a GUI facility. Session beans are used to implement business logic and do not represent persistent objects. They may be either stateless (and so having no memory of previous invocations) or stateful (and so capable of supporting a "conversation). Session beans may invoke entity beans in order to manipulate data in a database, but generally should not do so directly.

Container managed persistence causes access to entity beans, and therefore to the data they represent, to be automatically serialized. This avoids problems like deadlocks within the application server or, assuming only the application server accesses the data, within the database. While a policy of always using container managed persistence will avoid many database problems, it is important to note two potential issues. First, the automatically generated SQL tends to be record-oriented and to offer little opportunity for the sophistication of relational operators and the relational optimizer. Second, the technique is effectively application (server) controlled optimistic concurrency and can go horribly wrong if concurrent non-application server access to the database is permitted. Like extract processing, the data is checked out of or into the database via short transactions which do not hold locks while the application server is processing the data. To avoid potential concurrent access problems, it is advisable to keep the entity bean database separate from databases that are accessed by non-application server programs.

When bean managed persistence is used, much more sophisticated relational operations can be implemented by the developer. It also means that a developer who does not understand the relational model or how to write efficient SQL can create lots of problems. But then, that particular issue has been with us since long before the development of J2EE application servers, let alone WebSphere.

Conclusions

The WebSphere family of products extends beyond the application server and its IDE. WebSphere Business Integration provides a set of facilities for application integration including an integration broker, message queuing, business process modeling and simulation, and so on. WebSphere Commerce adds B2C and B2B capabilities. WebSphere Portal provides portal development and management for internal, partner, and customer portals.

WebSphere Application Server is perhaps the most widely used application server on the market. For most DB2 experts, it is very likely that some application in your environment will run under WebSphere Application Server. WebSphere uses DB2 for several purposes by default and integrates well enough with DB2 that Java developers really need not know very much about DB2 – or any relational DBMS – in order to use it. Of course, therein is the problem: WebSphere developers can just as easily abuse database access as use it. Space limitations have not permitted us to examine here all the potential interactions between WebSphere and DB2 that are important to DBAs, let alone many subtleties of J2EE. If for no other reason than to support the use community and protect the DB2 environment, DBAs need to have a general understanding of how WebSphere works.

Beyond these immediate concerns, IBM has invested a great deal in the WebSphere family of products and the underlying technology. There isn't space here to describe all the ways in which DB2 interacts with the entire WebSphere family, but almost all of these products require some form of data management and, for the time being at least, DB2 is the most common IBM solution. But the anticipated impact of J2EE application servers is even greater. IBM has stated that DB2 itself will become increasingly compatible with the WebSphere component architecture. For DB2 experts, I suggest that learning about WebSphere is not a choice but a survival mandate.

Note: This paper was originally published in IDUG Journal, Spring 2004.