Evaluating the Performance of EJB Components

An examination of two EJB-based architectures reveals differences that can significantly affect the performance and scalability of applications built on them.

As organizations do more business online, they need scalable, high-performance infrastructures to handle business transactions and provide access to core back-end systems. Middleware components help by providing mechanisms that make it relatively straightforward to build distributed applications. They provide services that support, for example, distributed transaction processing, security features, and directory services. Many middleware products are based on standard infrastructures such as Corba or the Java 2 Enterprise Edition (J2EE) or proprietary technologies such as COM+ or MQSeries.

As part of the Middleware Technology Evaluation (MTE) project (www.cmis.csiro.au/adsat/mte.htm), we conducted several experiments to explore the performance implications of two common application architectures supported by J2EE’s Enterprise JavaBean (EJB) component technology. One architecture promises simpler engineering and maintenance of the resulting component collection. For applications that require high performance and scalability, however, the alternative architecture might offer a better solution. Such knowledge is crucial to software architects, who must make initial design decisions early in a project, before extensive engineering has begun.

Enterprise JavaBean Architectures
Sun Microsystems’ J2EE is the standard for Java-based server applications. It defines a component-based approach to developing and deploying enterprise applications, making it possible to reuse Java components in a server-side infrastructure.

The EJB architecture specifies a programming model for constructing distributed object-oriented server-side applications in J2EE. A J2EE-compliant application server employs an EJB container to manage the execution of application components. Each container provides a process that hosts one or more
EJB components; when a client invokes a server component, the container automatically allocates a thread and invokes an instance of the component. The container manages resources on the component’s behalf and manages interactions between the component and the external systems. An EJB thus relies completely on its container and cannot execute independently.

Two common EJB application architectures are stateless session-bean-only and stateless session facade.2

A stateless session bean is an EJB that maintains no state on behalf of its clients. The session-bean-only architecture embeds the application business logic in a stateless session bean’s methods. The EJB container pools stateless session beans and shares instances between clients, which access the beans directly through the container’s interface. The session bean methods contain hand-coded Java Database Connectivity (JDBC) statements for accessing external data sources. Each session-bean method constitutes a single business transaction.

The stateless session facade uses a session bean as a wrapper for a set of container-managed persistence (CMP) entity beans, which encapsulate persistent data. With CMP, the EJB container generates the code necessary to map the entity bean data members to rows in a database. Clients call the session bean methods directly. Internally, the session bean methods call on the services of one or more CMP entity beans, which encapsulate the JDBC code for accessing external data sources. A single business transaction will thus span the stateless session-bean method and one or more entity beans that participate in the transaction.

Analysis of these two architectures reveals the following characteristics:

- **Ease of development.** The session-facade architecture is easier to develop because the EJB container generates the entity beans’ data access code, rather than requiring developers to manually code the JDBC calls.
- **Ease of maintenance.** Applications based on the session-facade architecture are generally easier to maintain because business logic changes typically require modifications to the session bean façade. Such changes are simple due to the architecture’s reliance on entity bean interfaces, rather than direct modifications of the more complex session-bean code and JDBC statements.

Analyzing the performance and scalability of these two architectures is somewhat more difficult. The session-bean-only alternative seems likely to perform and scale better because it uses (presumably) well-written and efficient JDBC statements, has only a single bean as a transaction participant, and maintains no state between client calls. In comparison, the session-facade alternative incurs more inter-EJB calls; has multiple transaction participants, slowing the commit protocol; and uses automatically generated JDBC statements that might not be optimal in terms of performance.

On the other hand, the EJB container can cache entity beans for use across transactions, potentially reducing the number of expensive database accesses and improving performance. Brebner and Ran explore these issues further.3

**Experiment Design**

Our study explores the performance and scalability trade-offs that result from deploying CMP entity beans in an EJB application. If entity beans can provide comparable or better performance and scalability than the session-bean-only architecture, then software architects have a compelling reason to use CMP entity beans. Other EJB architectural options exist, such as bean-managed persistence and stateful session beans, as well as non-EJB alternatives such as Java server pages or servlets accessing the database directly. These, however, are beyond the scope of this article.

**Simulated E-Commerce Application**

To investigate the performance implications of the two EJB application architectures, we used a sample application known as Stock-Online.4 The application simulates some typical e-commerce application functions and exercises the core features of an EJB container. It is more lightweight than other J2EE technology benchmarks, such as ECperf (http://java.sun.com/j2ee/ecperf/), which mandates the use of a single EJB application architecture. Stock-Online lets us experiment with application logic using different component architectures that exhibit different architectural trade-offs.

Essentially, Stock-Online simulates a simple online stockbroking system, letting subscribers buy and sell stock, inquire about current prices, and get holding statements detailing their stocks. Stock-Online’s server supports seven business transactions, from heavyweight update to lightweight read and update transactions. A database contains the minimum tables and fields for the system to sensibly operate.

For each product configuration we tested, the
number of clients varied from 100 to 1,000. For each test, clients performed 10 iterations of a fixed transaction mix, representing one complete business cycle at the client side. One iteration comprised 43 individual transactions, of which 81 percent were read-only and 19 percent were update transactions. Different transaction mixes can model different business cycle types. Ours aimed to represent a reasonably typical Web site load, where more than 80 percent of transactions are browse operations. The client loops through the transaction mix, selects a transaction type, performs the selected transaction, and attempts to commit it. When all iterations are complete, the client logs the performance measurements for postprocessing.

In some systems, a wait period, or think time, simulates the delay between a user’s transaction submissions. In Web-based systems with EJB backends, however, the interactive client code doesn’t call the business logic layer directly; rather, proxy client components launched by the Web server (JSPs or servlets, for example) do this. Because these proxy clients do not “think,” the tests do not have delays in the clients. Instead, the client immediately starts the next transaction upon completing the previous one.

**Architecture Implementations**

We implemented two versions of Stock-Online: a session-bean-only and a session-facade version. For each version, we deployed and tested the same code base on each of six J2EE application servers. These servers were the latest available production versions at the testing time.

- Borland Enterprise Server (BES) version 5.02
- Interstage Application Server (IAS) version 4.0
- SilverStream Application Server (SS) version 3.7.4
- WebLogic Server (WLS) version 6.1
- WebSphere Application Server (WAS) version 4.0
- JBoss version 2.4.3

The session-bean-only version uses a single session bean that supports Stock-Online’s seven business methods, which access the corresponding database tables using JDBC statements. In the session-facade version, the client applications access the interface provided by the stateless session bean, but we incorporated four entity beans into the application architecture that map their data members to data in one of the database tables. The session bean uses the entity beans to perform the access logic, and the entity bean writes its data members to the corresponding table and columns, defined using product-specific deployment tools.

Both versions use container-managed transactions. Thus, the EJB container manages a transaction’s creation and coordinates its outcome, requiring no application code.

**Test Environment**

Our test environment used four Dell PowerEdge 6450/700 machines running Windows 2000. Each had four Xeon 700-MHz CPUs and four Gbytes of memory.

The database ran on another Windows 2000 machine (an IBM Server 8681) with eight CPUs, which was sufficiently powerful to ensure that the database was not the bottleneck in the performance tests. In fact, the load on the database machine was typically from 10 to 30 percent. We implemented the database using Oracle version 8.1.5.

An isolated 100-Mbit switched Ethernet ran...
between test machines. The test configuration deployed the simulated clients, application servers, and database on physically different machines.

We followed the vendors’ performance tuning recommendations for all products and executed the experiments using various configuration settings to get the best performance in the test environment. This was a time-consuming (and, with some products, difficult) process that required extensive vendor support. The test code obtained average client response time for each transaction and application throughput in terms of transactions per second, which it logged to file at the end of each test.

Experimental Results

For both the session-bean-only and session-facade architectures, we ran two sets of tests for each of the six J2EE servers. The first set executed the EJB components on a single machine. The second set executed a clustered configuration in which two physically separate servers executed the EJB components. Because JBoss version 2.4.3 does not support clustering, we didn’t test it in a clustered configuration.

BES exhibits the greatest throughput in nearly all tested configurations. For example, in the clustered session-bean-only and session-facade tests, it sustains more than 3,500 and 2,100 transactions per second, respectively, across all client loads. To condense the test results presentation, we normalized each application server’s measured throughput to the Borland server’s performance.

Figure 1 compares the results from the two tests for all six technologies using the session-bean-only architecture. The graphs show the normalized application throughput with client loads varying from 100 to 1,000.

Although all six application servers implemented the same component standard and executed the same code in the same test environment, significant differences emerged between their raw performances. Those with lowest throughput provide between 16 and 27 percent of BES’s performance on a single server, and between 21 and 30 percent of BES in a cluster.

Figure 2 compares the normalized performance for the single-server and cluster configurations using the session-facade architecture. We found a problem with SilverStream’s entity bean implementation, and hence could not measure its performance. The difference in absolute performance between products is again significant. In the single-server test, JBoss provided around 10 percent of BES’s performance for up to 500 clients; with 1,000 clients, it failed to complete the test. In a cluster, IAS outperforms BES for all tests up to 1,000 clients. WAS and WLS are relatively stable in both test configurations, providing between 60 and 75 percent of BES’s performance.

Figure 3 (next page) depicts each application server’s scalability for the two architectures. Ideally, the cluster configuration should provide 100 percent scalability over the single server, as the cluster has double the processing resources available, but this is clearly not the case. All but IAS show scalability from 60 to 90 percent across all client loads, indicating overheads in the clustering mechanisms that inhibit scalability. Interestingly, IAS shows greater than 100 percent scalability for nearly all client loads. This is because IAS quickly saturates the resources on a single machine, causing performance to drop as demand for scarce resources increases contention. By clustering, IAS effectively balances the client load over the two machines, and this greatly reduces...
resource contention. Hence the overall application performance benefits significantly.

For both architectures, as the client load increases and the application’s processing capacity remains constant, the applications should ideally exhibit constant throughput, with a corresponding linear rise in individual client response time. Thus, in our experiments, the throughput observed with 100 clients should not differ from that observed with 1,000 clients. As Table 1 shows, only BES achieves this goal in all tests. In general, the cluster configurations show more scalable characteristics, indicating that the application servers can exploit the additional processing resources.

Finally, we compared product performance from the session-bean-only and session-facade tests. Table 2 compares each product’s highest and lowest performance across all client loads for both the single-server and cluster configurations. Three application servers demonstrate similar characteristics, with the session-bean-only architecture ranging from 1.37 to 1.74 times faster than the session facade architecture across both configurations. JBoss and IAS, however, exhibit different characteristics. JBoss’s session facade implementation is more than four times slower than its session-bean-only implementation, whereas the session facade architecture provides greater performance for IAS.

As Table 3 shows, we obtain another perspective on the data by ignoring specific products and comparing the highest session-bean-only performance at a given client load with the highest and lowest session facade performance.

These results emphasize the importance of product and architecture alternative selection for applications that require high performance and scalability. In the worst-case scenario, appropriate technologies and architecture designs can produce better than a 20-fold increase in performance for the same application.
Implications
The results of this work have implications for both software architects and researchers. For software architects, our results emphasize the importance of product and architecture selection. Because both can profoundly affect the resulting application’s performance and scalability, organizations should exercise due diligence in selecting a technology. J2EE product-release cycles — typically from 12 to 18 months — exacerbate this situation as new versions might, or might not, dramatically improve product performance. Further, architects should understand the implications of the architectural trade-offs inherent in their designs. For applications in which high performance is crucial, prudent use of prototyping should confirm that a given architecture can provide the required performance levels.

For researchers, the ability to compose components into applications with predictable behavior is an enticing goal. In this context, one aim would be to accurately predict the application’s performance without having to build and test the application. Our work shows how component infrastructures such as EJB actually magnify this problem by complicating performance prediction. Such technologies tightly couple the component infrastructure and application components, so you cannot divorce the performance of one from the other. As the results show, different EJB implementations vary greatly in performance. Hence, useful performance-prediction techniques must take product-specific issues into account.

Several researchers are performing interesting analyses of component technologies. This work typically relies on the component container’s source code being available for instrumentation, making white-box analysis of the container behavior possible. Clearly this assumption is not valid for most commercially available component technologies, which must be treated as black boxes.

References

Ian Gorton is chief architect of information sciences and engineering at the US Department of Energy’s Pacific Northwest National Laboratory. His research interests include software architectures. Gorton received a PhD in computer science from Sheffield Hallam University. Contact him at ian.gorton@pnl.gov.

Anna Liu is an enterprise architect with Microsoft Australia. Her research interests include software architectures and COTS software-evaluation and acquisition methods. Liu holds a PhD in computer engineering from the University of New South Wales, Australia. Contact her at annali@microsoft.com.