Designing WebSphere Application Server for performance: An evolutionary approach

by R. Willenborg
K. Brown
G. Cuomo

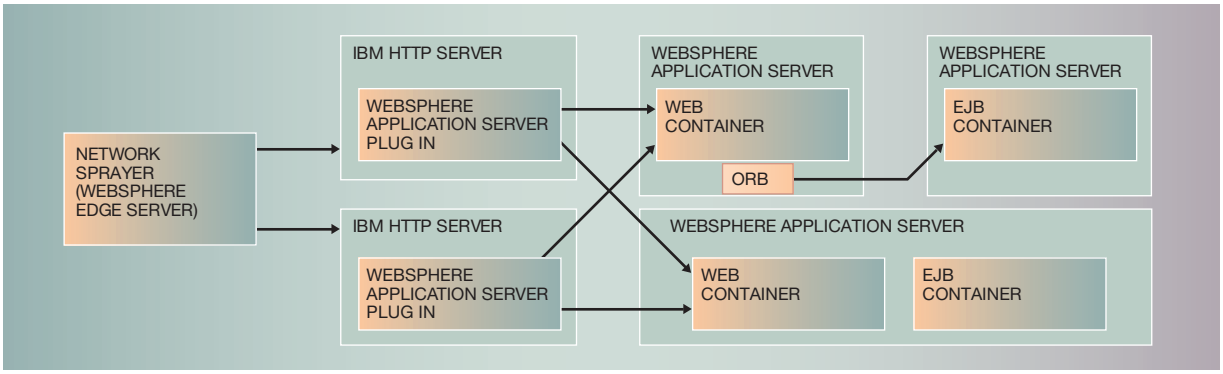
Performance and scalability are critical elements for a successful Web site and therefore, are fundamental design criteria for the IBM WebSphere® platform. This paper focuses on the evolution of IBM WebSphere Application Server performance and scalability features, the improvements that were achieved, and directions for future work. There are three design principles underlying the success of WebSphere: (1) optimize the design for the predominant (no-failure) case, (2) make finite resources appear infinite, and (3) minimize the number of interprocess calls. We illustrate the application of these design principles in key areas, including workload management, Hypertext Transfer Protocol (HTTP) session management, back-end connection management, session data and content caching, and Enterprise JavaBean™ (EJB™) design and deployment patterns.

WebSphere Application Server is IBM's Java-based Web application server that supports the deployment and management of Web applications, ranging from simple Web sites to powerful e-business solutions.[1] WebSphere Application Server performance, and in particular its scalability and resiliency, have consistently improved from release to release. We describe here our evolutionary approach to improving the performance of WebSphere Application Server and the design principles we adopted that enabled these performance improvements.

WebSphere Application Server is, at its heart, a Java 2 Platform, Enterprise Edition (J2EE**)[2] Web application server, similar to a number of other Web application servers, such as BEA WebLogic** Server[3] and Oracle Application Server.[4] J2EE is a platform for building distributed enterprise applications that includes a specification, a reference implementation, and a set of testing suites. Its core components are Enterprise JavaBeans** (EJBs**), JavaServer Pages** (JSPs**), Java** servlets, and a variety of interfaces for linking to databases and other information resources in the enterprise. The components of various types are deployed in "containers" that provide services for those components: servlets are deployed into a Web container, whereas EJBs are deployed into an EJB container.

Most Web applications written for J2EE have a common architecture, commonly referred to as the model-view-controller (MVC) architecture.[5] Its main advantage, which is the reason for its widespread adoption, is the separation of design concerns: data persistence and behavior, presentation, and control. Thus, control is centralized, code duplication is decreased, and changes of code are more localized. For example, changes to the presentation (view) of the data are limited to the JSP components, whereas changes to the business logic (model) are limited to the business model components. The controller, usually implemented as a Java servlet and associated classes, mediates between the view and the model and coordinates application flow.

Figure 1 Major components in a WebSphere Application Server installation



The challenge to vendors of J2EE application servers is to support the deployment of applications that serve hundreds or thousands of simultaneous users. Such a load is greater than any single machine can handle. Moreover, because a single machine is susceptible to hardware failure, the design of such systems has to include failover. Failover is a backup mode in which the functions of a system component (e.g., a processor, network, or database) are assumed by other components when the primary component becomes unavailable through either failure or scheduled down time. The design also has to support the administration of multiple servers, as well as the management of workload and performance across these servers.

In our work to improve application performance measures we have used three main design principles, or themes: (1) optimize performance for failure-free, normal operation (that is, treat failover as a special case), (2) make finite resources appear infinite, and (3) minimize cross-process calls. While this paper is primarily a historical overview of the evolution of these major themes in the product, we expect that developers of WebSphere Application Server applications will gain additional insight into the features of WebSphere Application Server and will thus be able to optimize their applications. We also expect that readers interested in distributed systems will gain some insight into the way the three design principles have been used in WebSphere Application Server and will be able to apply some of the lessons we learned to their own systems.

The rest of the paper consists of three main sections, each of which resonates with one of these three themes. In the next section, we examine the evolution of WebSphere Application Server scalability and resiliency (a.k.a. availability). This section is based on the principle "optimize for failure-free operation" as applied to workload management and data partitioning. In the following section, in which we examine the evolution of resource management, we apply the principle "make finite resources appear infinite." In this section, we also describe the interplay between application performance and the WebSphere Application Server infrastructure, and the role WebSphere Application Server has played in providing high-performing standard interfaces for application developers. The principle "minimize cross-process calls" anchors the section that follows, in which we cover the evolution of caching and EJBs. We continue to examine the relationship between the WebSphere Application Server infrastructure and the application, discussing application design and deployment topology. We also discuss the unique content-caching capabilities of WebSphere Application Server that go well beyond the existing J2EE specification. We conclude with a brief summary.

The evolution of scalability and resiliency: Optimize for normal processing

Figure 1 shows the major components in a WebSphere Application Server installation: network sprayer, Web server, WebSphere Application Server, Web container, EJB container, WebSphere Application Server plug-in, and ORB (Object Request Broker).[6] The workload, consisting of client HTTP (HyperText Transfer Protocol) requests, is routed through these components. Each client request is eventually mapped to an execution thread that pro-

cesses this request. As shown in Figure 1, there are three major routing points in WebSphere Application Server: the network sprayer, the WebSphere Application Server plug-in, and the ORB.[7]

Figure 1 shows two deployment options for Web containers and EJB containers. Each container can be deployed by itself within an application server (the two application servers located in the upper part in Figure 1), or they can be co-deployed within a single application server (the application server in the bottom part of Figure 1).

The network sprayer routes the arriving HTTP request to a Web server. The Web server is "WebSphere Application Server-enabled"; that is, it is equipped with a WebSphere Application Server plug-in that forwards requests from the Web server to a WebSphere Application Server. Because of its strategic positioning as the first point of WebSphere Application Server presence in the installation, the plug-in has built-in functions for workload management, security, and caching.

The ORB, the third routing point in our typical installation, routes calls to EJB methods to an application server that hosts an EJB container. The ORB also supports failover by resending failed requests to another application server. The ORB makes use of the Internet InterORB Protocol (IIOP**), which has the advantage that requests not originating in a Web browser can still benefit from workload management.[7]

We describe now our approach to enhancing the scalability and resiliency of the system in Figure 1. As we will show, the techniques used to make a system scalable often make the system resilient (highly available) as well. Over the past three releases of WebSphere Application Server, a pattern has emerged that has become our blueprint for enhancing the scalability and resiliency of the product. Our approach involves the use of the following techniques: clustering, workload management, data partitioning, caching, and data replication.

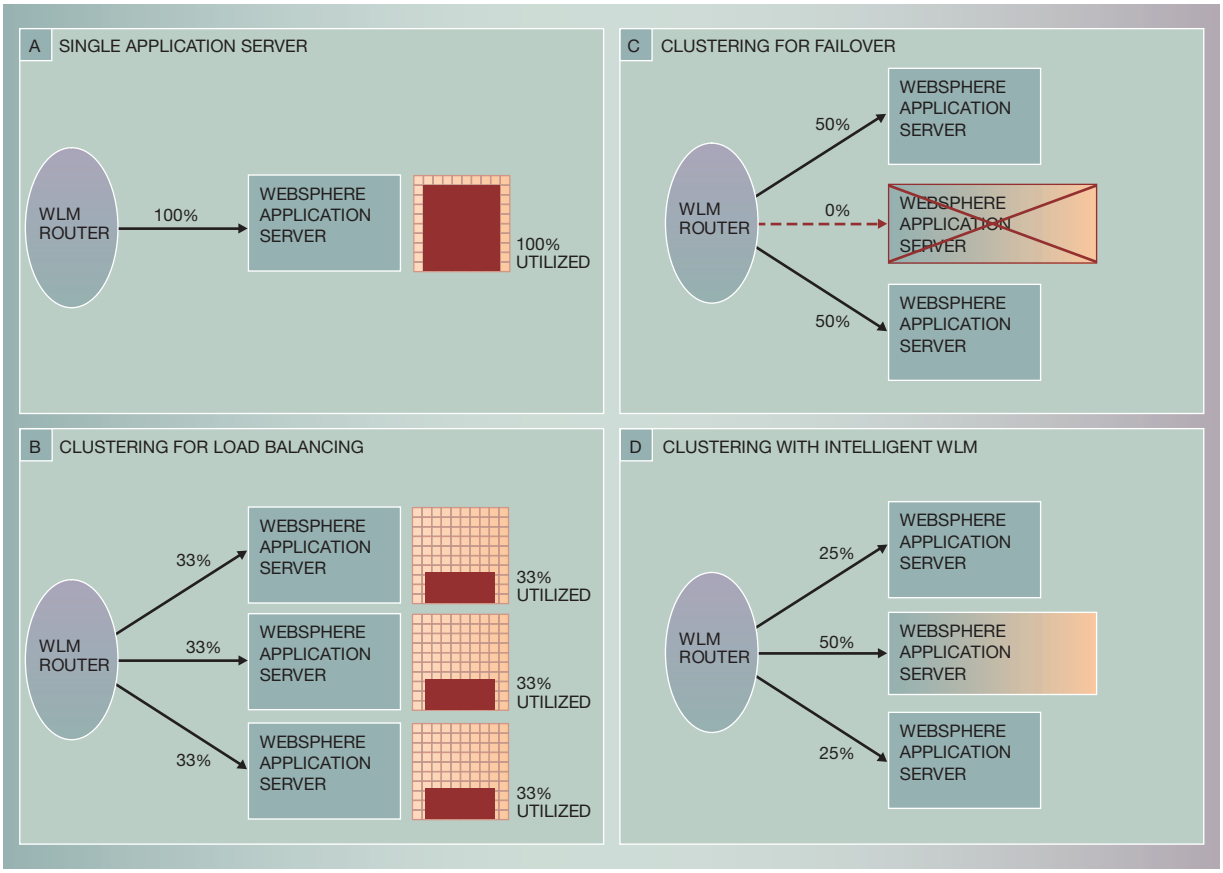
Clustering. Our primary technique for achieving scalability and resiliency is clustering. (See Figure 2.) When a single application server cannot support a site's performance requirements, we can obtain significant improvements in application throughput and response time by running multiple copies of an application on a cluster of application servers (Figure 2B). If the application is well-written and the WebSphere Application Server system is properly configured and provisioned, then close to linear scalability can be achieved. Figure 2C illustrates a failover scenario, in which a failure of one of the nodes in the three-node cluster is handled by having the remaining nodes take over the entire load.

The scalability of WebSphere Application Server through clustering has been demonstrated in numerous benchmarks and customer engagements over the years. The ECperf** Expert Group, for example, publishes a number of results of measuring performance and scalability of J2EE servers.[8] The study of April, 22, 2002, shown in Figure 3, reflects measurements involving a cluster of up to nine nodes, using WebSphere Application Server Version 4.03, and a common database consisting of IBM DB2 Version 7.2. As Figure 3 illustrates, the experiment demonstrated near linear scalability for the throughput (a multiplier of 8.74 corresponds to a nine-node cluster). The graph marked "DB2 CPU" shows the utilization of the database server as a function of the cluster size. The unit of measurement is in BBOPS (Benchmark Business Operations). See Reference 9 for additional benchmarking results for application servers, including WebSphere Application Server. The ORB workload routing capabilities of WebSphere Application Server are not duplicated by other application servers in these tests.[10] As these results show, WebSphere Application Server has been at or near the top among application servers in terms of performance and scalability. (See also Reference 11.)

Workload management. In a cluster of application servers, the technique used for distributing the workload among the application server instances is called workload management (WLM). The administrator defines policies that dictate how requests are distributed to the applications running in the cluster (this is known as intelligent workload management). WLM routers (network sprayer, WebSphere Application Server plug-in, ORB) interpret these policies and route requests to downstream WebSphere Application Servers. Figure 2D illustrates the distribution of workload in a three-node cluster, where the work is split 25 percent, 50 percent, and 25 percent.

Data partitioning. Early in the development of application servers it could be assumed that no client data need be stored from one session to the next ("stateless" design). Applications could thus be made to scale linearly with the size of the cluster through the use of simple workload management policies. Following this approach, early WebSphere Appli-

Figure 2 Clustering Scenarios



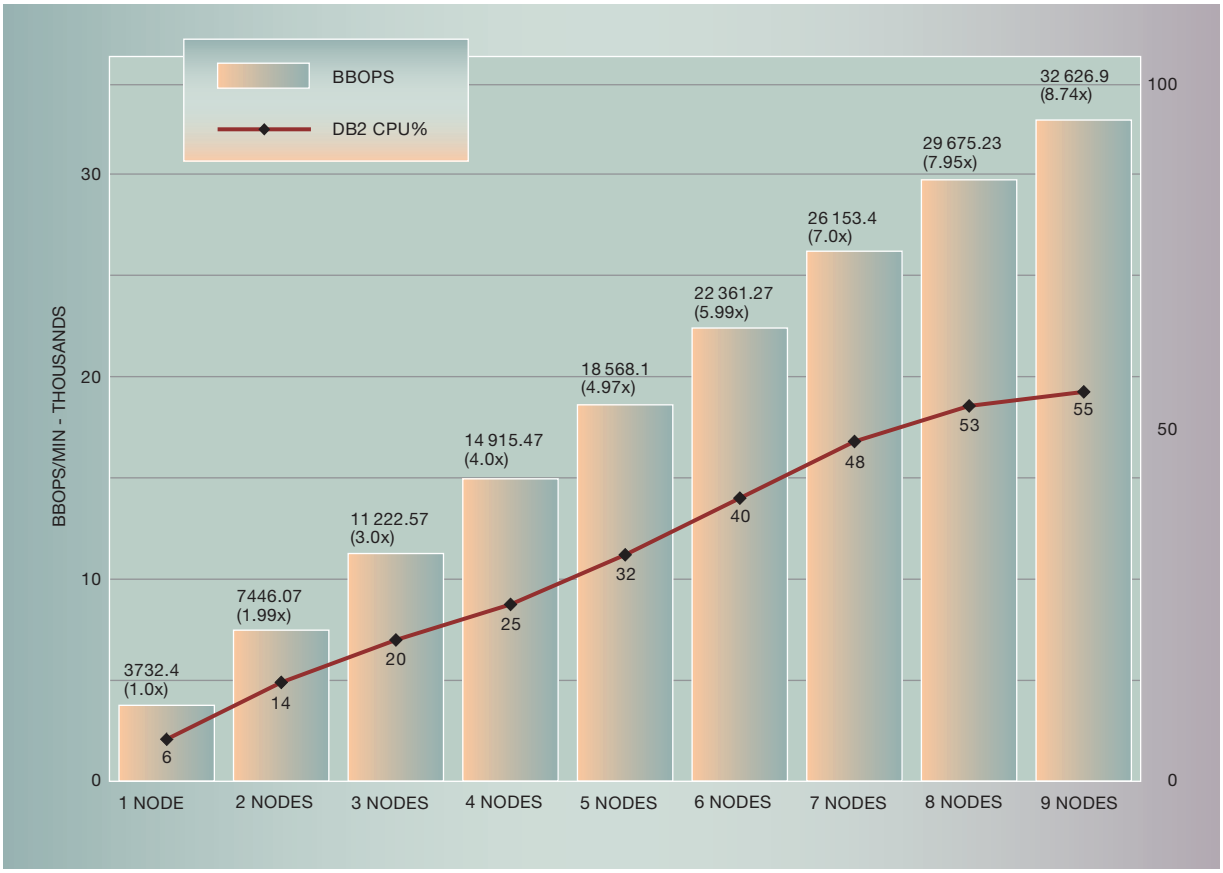
cation Server customers of high-volume performance-sensitive applications would go to great lengths to develop applications that were designed to be stateless. Although linear scalability was achieved, the benefits were often short-lived.

As applications began providing richer and personalized user experiences, session state data had to be stored on the application server. At first, the arriving transactions could be routed to any server, and thus session state data had to be accessible by all servers in a cluster, as illustrated in Figure 4A. This was typically accomplished by storing the data on shared persistent storage, such as a database. This approach suffers from performance degradations due to the significant data transfers involved and the overhead of keeping track of the most recent update of a data item when there is more than one copy of it in use (all copies of a data item other than the most recent one have to be invalidated and cannot be reused).

To improve performance for "stateful" applications we introduced data partitioning. The technique is illustrated in Figure 4B for the no-failure case (the possibility that one of the nodes in the cluster fails will be considered later). Session state data is partitioned four ways, each partition is resident on a designated server. The database component is not needed for the no-failure case because the session data is stored on the designated server (we refer to this as data "affinity"). This data affinity is the basis for intelligent workload routing, which results in reduced traffic and overhead and improved performance.

Although the combination of data partitioning and intelligent workload management improves the performance in a cluster, the configuration has a single point of failure. We address this deficiency in the next section, which deals with data replication.

Figure 3 ECPerf scalability results for WebSphere Application Server Version 4.03, 04/22/2002



Data replication and caching. To improve on the configuration in Figure 4B and provide for failover, we use a database for storing data as illustrated in Figure 5. We also use data partitioning and establish an affinity between each partition and a selected cluster node (Figure 5 depicts the partitions in different colors). The backing of each data partition to the database follows a write-through caching mechanism, in which any changes to the data in the cluster node are immediately replicated on the database. In the case of a server failure, the WLM router detects the failure and selects another server to which the data partition associated with the failed server is assigned. When the first client request after the failure arrives at the new server, the user data is transferred from the database to populate the new cache for this data partition.

The evolution of HTTP user session support. The use of the previously described techniques for enhancing scalability and resiliency is best illustrated in the WebSphere Application Server support of HTTP user sessions. Table 1 summarizes the release-to-release evolution of the use of these techniques.

WebSphere Application Server Version 3.0. The first WebSphere Application Server implementations typically relied on the network sprayer for workload management across multiple HTTP servers and WebSphere Application Server instances. For stateful applications, the network sprayer "sticky port" option was used, which routed all requests from a given Internet Protocol (IP) address to the same HTTP server. While this simplistic form of workload management worked for some Web sites, the workload was not always well distributed. For example, all users coming through a proxy server IP address were routed to the same Web server. In particular, large Internet service providers with many users coming through a single proxy address caused unbalanced

Figure 4 Workload routing with and without data partitioning

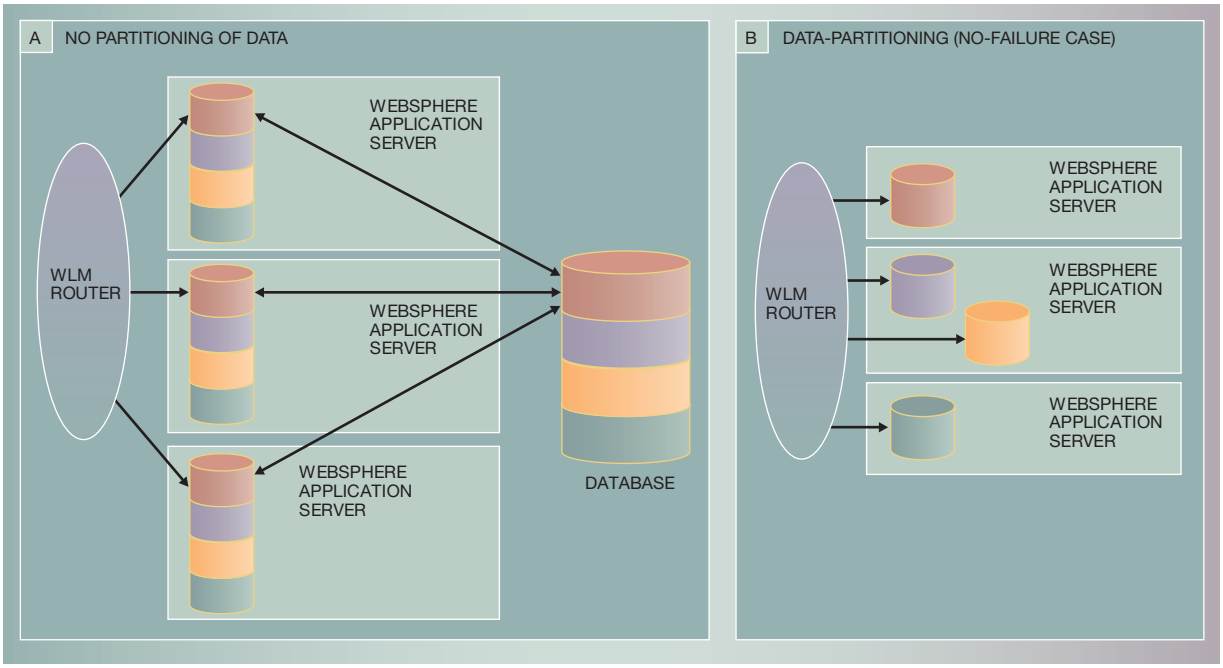


Table 1 Evolution of HTTP user session support

	WebSphere Application Server Version 3.0	WebSphere Application Server Version 3.5	WebSphere Application Server Version 4.0	WebSphere Application Server Version 5.0	WebSphere Application Server Enterprise Edition Version 5.02
Workload Management	Network sprayer sticky port	WebSphere Application Server plug-in affinity from second request	WebSphere Application Server plug-in pure affinity	WebSphere Application Server plug-in / ORB weighted workload management	Performance PMI-based dynamic workload management
Caching	No session caching	Write-through session cache	Lazy session cache	Lazy session cache	Lazy session cache
Replication Store	Relational database	Relational database	Relational database	Relational database, memory	Relational database, memory

workload distribution. An even worse situation was created when requests from the same end user came through two different proxy servers and thus gave the appearance of two distinct users, which interfered with session data partitioning schemes.

WebSphere Application Server Version 3.5. Clearly, simple workload management at the network sprayer was not sufficient. Thus, in WebSphere Application Server Version 3.5, affinity-based routing was added at the WebSphere Application Server plug-in layer. By default, client requests were routed using simple routing patterns, such as round robin or random. However, when a user session was detected, the routing mechanism identified the cluster node where the user session was cached by querying the data partitioning component and forwarded the request to it.

The first client request triggered the creation of a session identifier (session ID) and an associated HTTP cookie at the client. A second request triggered the creation of an affinity of the user session to a data partition (and thus to a cluster node) by hashing the session ID. Although the hashing approach minimized the amount of data required to determine the partition (it needed just a simple session ID), it was not totally effective because affinity was not established until the second request. As a result, the first request was sometimes routed to a server different from the second and subsequent requests. Thus, under normal operation (no failure), session data had to be staged from the database not once, but twice.

As illustrated in Figure 6, the same user session is served by two WebSphere Application Servers. Session data is created following the first client request (label 1 in the figure). When the transaction processing is complete (end of service or EOS), session data is written through to the database. In other words, WebSphere Application Server Version 3.5 does not cleanly partition HTTP session data. Second and subsequent client requests (label 2-n in the figure) are routed to another WebSphere Application Server, and session data is staged from the database. These additional database accesses hinder performance, especially for large HTTP sessions.

WebSphere Application Server Version 4.0. This release recognized a critical tenet for achieving high performance and scalability—optimize for normal path processing and treat failures as a special case. In other words, for performance and scalability, assume failures do not happen! We still support failover through techniques such as data replication, but we remove failover-related operations from mainstream processing.

Thus, in WebSphere Application Server Version 4.0, access to the database is required only in failover scenarios. At the time the session ID is created, a server ID, which establishes an affinity between the session and a specific server, is also created. This ensures that this session's requests are all routed to the same WebSphere Application Server, a property known as "pure affinity" (the system illustrated in Figure 5 exhibits this property).

With this addition, the session data is always available in the server memory, and access to the database is needed only if a failure occurs. Further performance optimizations were added in the data replication layer with a customizable lazy replication

Figure 5 Data replication for failover

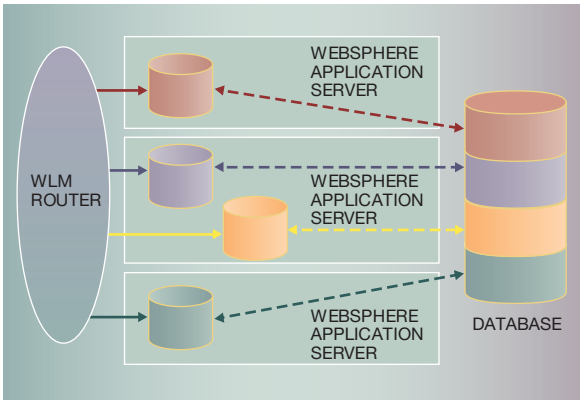
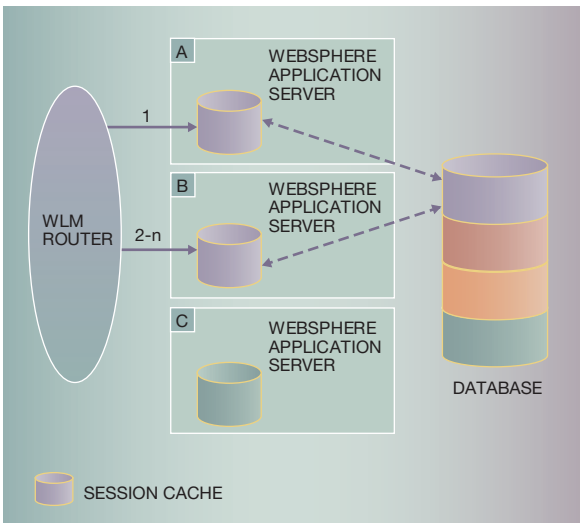


Figure 6 WebSphere Application Server Version 3.5 session affinity



feature. Lazy replication, unlike the EOS alternative, allows the updating of session data to the database to be queued up and processed by a background thread. This feature, referred to as time-based write (TBW), allows trading off performance for robustness by varying the time delay in triggering the data replication task; the longer this time delay the higher the probability that session data will be lost in case of failure.

The performance advantages of lazy replication in WebSphere Application Server are quite dramatic,

Figure 7 Performance of EOS versus TBW database replication policies

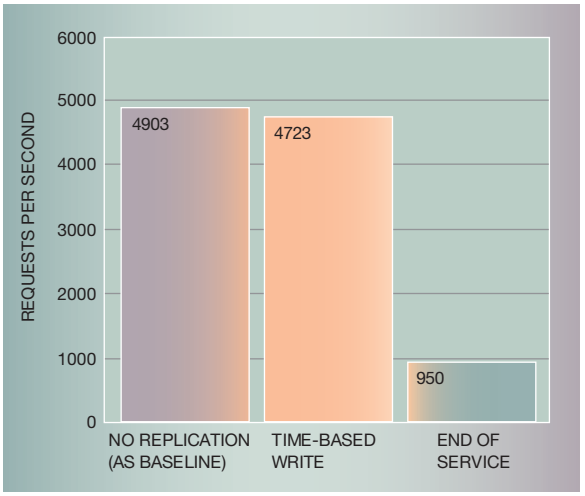
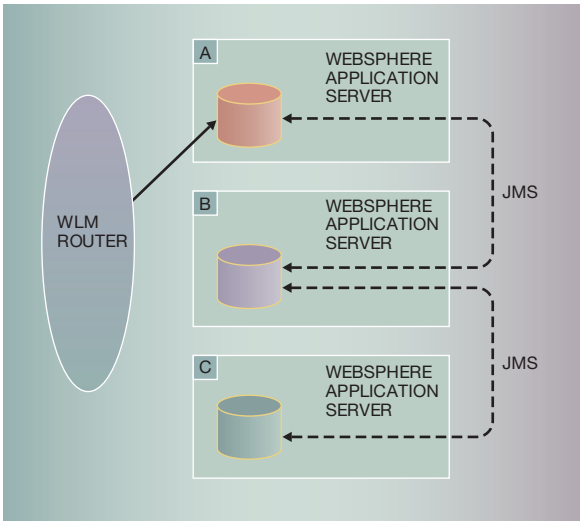


Figure 8 WebSphere Application Server Version 5.0 memory-to-memory session data replication



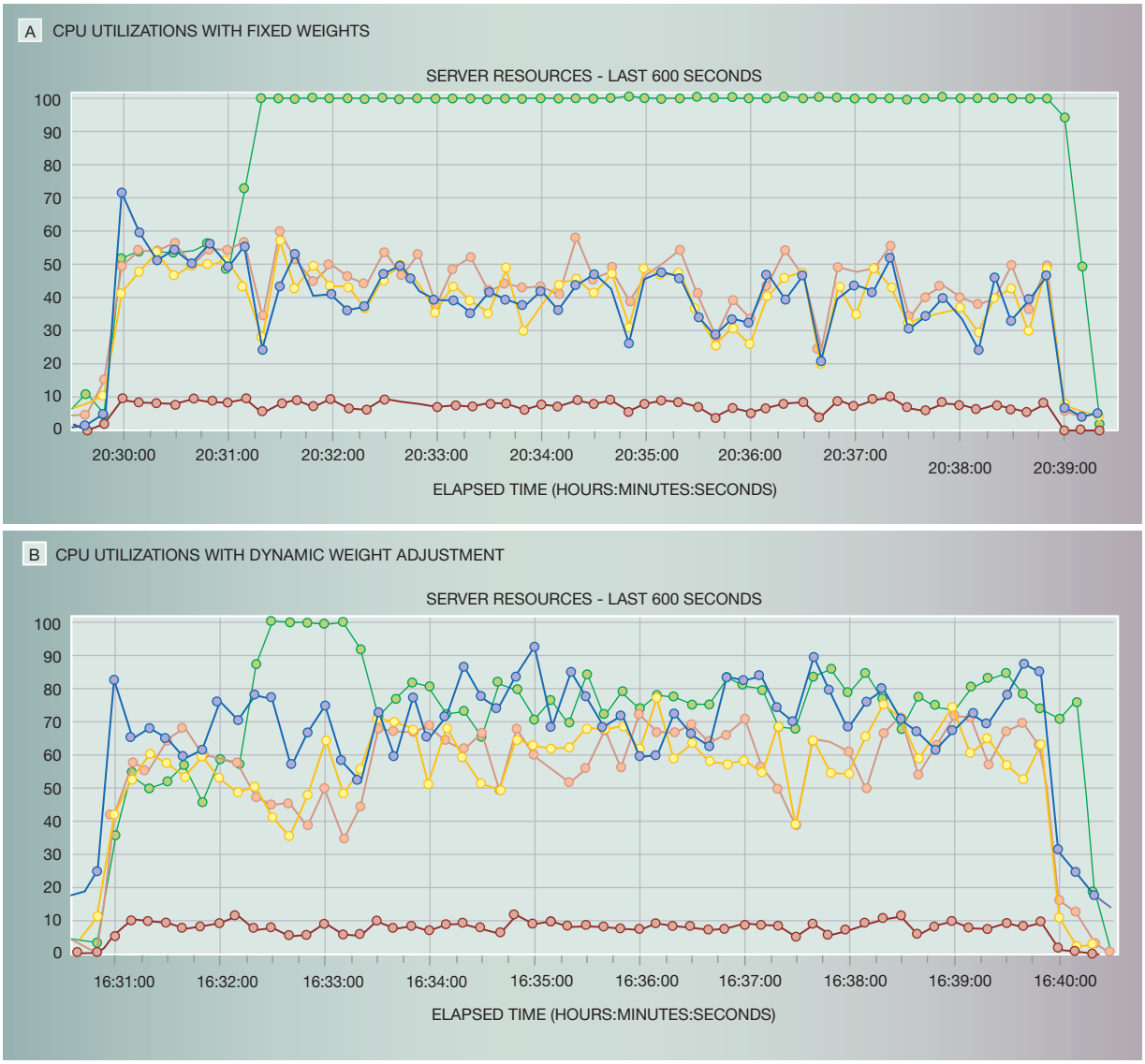
particularly for larger session objects. Figure 7 shows the performance of EOS versus time-based write-session data-replication policies for a simple servlet test case with a 2 KB HTTP session object.[12] This test case involved a simple performance primitive specifically designed to study the performance of session replication and contained no business logic. The bar graph shows the same test case executed in three different configurations. First, the baseline is run without replication of session data to the database, which represents the "best case" performance. The same test is then executed with database updating using a TBW delay of 10 seconds, and then with EOS processing. The performance degradation is small (at 4 percent) with TBW, but significant with EOS (at 80 percent). Although this simple test case illustrates the potential for performance improvement with TBW, a typical Web application will experience more modest benefits. The WebSphere Performance Benchmark Sample, for example, shows comparable degradations of 1 percent and 18 percent respectively.[12]

WebSphere Application Server Version 5.0. Enhancements for this release were focused on data replication and workload management. With pure affinity, HTTP session replication to a database was only required for failover scenarios. Although this means of data replication works well, it assumes that the database server is highly available. Unless the installation already includes a highly available database, this can be an expensive proposition.

As an alternative, WebSphere Application Server Version 5.0 introduced *memory-to-memory replication*. As shown in Figure 8, servers in the cluster are configured to act as back-up servers for user-session data. This configuration backing up the session data on a given server is known as its *replication domain*. The replication domain can consist of a single server (buddy) or a collection of servers (in Figure 8, server B is server A's buddy). The memory-to-memory replication of session data is performed using the built-in Java message broker, Java Messaging Service (JMS).[2] This solution provides a cost effective means of scaling WebSphere Application Server user sessions when a highly available database is not already available because it uses the existing WebSphere Application Server infrastructure (no need for an additional database server).

In addition to the new data replication capabilities, Version 5.0 included an enhanced workload routing mechanism in which the WebSphere Application Server plug-in and ORB components had the ability to perform weighted round-robin routing. For example, if in a two-node WebSphere Application Server cluster, the first node is a 1GHz (gigahertz) 2-way symmetric multiprocessor (SMP) and the other a 1 GHz 8-way SMP, we may want to route four times as many requests to the 8-way machine in order to ensure the machines are properly utilized. With

Figure 9 Dynamic workload management performance results



weighted round-robin routing, this is done by the appropriate setting of the weights in the routing algorithm.

WebSphere Application Server Enterprise Edition Version 5.02. Workload management in this release was enhanced with the addition of *dynamic workload management*, which extends weighted round-robin routing by automatically adjusting the weights based on performance metrics, such as response time and throughput. The performance data is obtained from the WebSphere Application Server Performance Monitoring Infrastructure (PMI).[1]

The result of dynamic workload management is a more even distribution of work across a cluster, typically resulting in higher overall throughput when application usage patterns change. Figure 9 shows performance results from a test environment with four nodes running two applications; Figure 9A shows the fixed-weights case; whereas, Figure 9B shows the results with dynamic weight adjustment.[13] Application

1 is deployed on all four nodes; whereas, application 2 is deployed on a single node. For the first 1-minute interval in Figure 9A, in which only application 1 is active, the CPU utilization across all four servers is comparable and running around 50 percent. As a second application is activated, the fixed-weight routing algorithm continues to evenly distribute application-1 requests to the four nodes in the cluster. As a result, the CPU on one server reaches 100 percent, while the other three servers remain at around 50 percent utilization.

In contrast, Figure 9B shows the result of using dynamic workload management with the same workload mix. In this case the system automatically adjusts to CPU utilization, so that additional application-1 requests are routed to the three underutilized servers. As a result, the CPU utilization on these three servers increases to the 60-70 percent range (at time 16:34, stability is reached); whereas, the server that was previously running at maximum capacity now operates in a similar range. Moreover, the throughput for application 1 increases by 16 percent.

Future directions in scalability and resiliency. Achieving scalability and resiliency through clustering, data partitioning, caching, intelligent workload management, and data replication extends beyond HTTP transaction processing. Current implementations of stateful session beans and Option-A Cached CMP (container-managed persistence) entity beans[2] are limited in clustered environments. Unlike an HTTP session, the infrastructure does not have the application knowledge to automatically perform the data-partitioning and intelligent-workload management. New infrastructure services, combined with application design and implementation techniques, are being explored to improve performance and scalability for stateful session beans and Option-A Cached CMP entity beans. The same design principle, optimized for normal (no-failure) path processing, applies.

Although WebSphere Application Server is optimized for the no-failure case, mission-critical requirements are driving investigation into more aggressive ways of handling failures. WebSphere Application Server currently supports a passive failure detection scheme, as follows. When a server in a cluster goes down, the upstream WLM agent (i.e., the WebSphere Application Server plug-in or the ORB) detects the failure when it attempts to connect to the failed server. The failure detection mechanism involves time-out and retry intervals, and these represent delays in responding to the failure. We are working on more powerful failure detection schemes that include a heartbeat mechanism and that allow timely detection of failures.

The evolution of resource management: Make finite resources appear infinite

In addition to designing a scalable and resilient WebSphere Application Server infrastructure, we also need to manage available resources. Memory, CPU, and disk storage are typically the constraining resources in any software environment. WebSphere Application Server, which runs in a Java Virtual Machine (JVM**), is additionally constrained by the resources made available to the JVM by the operating system.

As a server-side Java operating environment, WebSphere Application Server must utilize these resources wisely and not impose resource restrictions on the applications it hosts. These applications have widely diverse resource requirements. Whereas some applications handle high-volume short-transaction traffic, others have long running times and require large amounts of memory. Many WebSphere Application Server applications were ported from a client/server environment where resource constraints are less severe than in distributed multitiered environments.

Regardless of the application mix, the WebSphere Application Server environment typically hosts many simultaneous requests, each of which requires resources. Because the resources required at any one time often surpass the available resources, WebSphere Application Server is designed to make the finite resources appear to the application as unlimited. A variety of techniques, such as queues, thread pools, and memory utilization schemes, have evolved to support the two most critical resources: threads and memory.

In this section we examine the evolution of resource management in two key areas: access to databases and other back-end resources, and thread pool dynamics. This evolution has been driven by the need for standards and by requirements for performance, scalability, and security. Because performance and scalability require careful control of resources, the WebSphere Application Server designers faced the challenge of providing high performance infrastructure components that did not impose explicit limits

Table 2 Evolution of connection management

WebSphere Application Server Version 3.0	WebSphere Application Server Version 3.5	WebSphere Application Server Version 4.0	WebSphere Application Server Version 5.0
Database connection pooling (proprietary)	Pre-standard JDBC 2.0 connection pooling	JDBC 2.0 standard connection pooling; proprietary support for CCF	J2EE Connector Architecture (J2CA) standard replaces CCF

on application developers. Their solution was making the finite resources appear infinite.

Access to database and other back-end resources. Although the middle tier of the WebSphere Application Server environment, which includes the interaction between the Web servers and the WebSphere Application Servers, has its own challenges, the back-end tier has been the most challenging one. This back-end tier is not considered part of the WebSphere Application Server system, and it is often maintained, tuned, and administered separately from the WebSphere Application Server system. Almost all applications require data from the back end, which includes either a database or a "legacy" system. Back-end resources are usually constrained, often with hard configuration limits. We need to develop applications without explicit knowledge of these restrictions. Moreover, providing a mechanism to make these finite back-end resources appear unlimited to the application is important for both performance and ease of development.

From a performance perspective, the creation of back-end connections is often extremely expensive. Many performance bottlenecks are created when the application accesses the back-end tier, typically a relational database. These databases can be accessed either directly by the application using Java Database Connectivity (JDBC**),[2] or by WebSphere Application Server CMP EJBs. We examine now the evolution of database access (database connection management) within a WebSphere Application Server system, followed by more general capabilities for accessing the back end. This evolution is summarized in Table 2.

WebSphere Application Server Version 3.0. The earliest WebSphere Application Server applications were supported by servlets accessing a relational database. The performance of these applications was heavily dependent on database access. The most performance-costly JDBC command is getConnection. Establishing the connection may take seconds, much longer than typical SQL (structured query language) queries over JDBC.

To minimize the number of getConnection calls required, a common technique is to use a connection pool. Though this concept was not new, at the time there was no connection pool support in the JDBC 1.0 specification. The WebSphere Application Server designers recognized the importance of optimizing database access and provided a proprietary database connection pool service for use by application developers.

This database connection pool service maintains a set of connections, making these connections available to the application as needed. Thus, rather than creating and destroying connections, the connections in the pool are allocated on request and returned to the pool for reuse. Though the number of connections is finite, the connection pooling service makes them appear unlimited to the applications. The performance overhead of the getConnection call is never experienced by the running application. This simple concept is highly efficient, and the use of connection pooling yielded significant performance improvements. Gunther documents these performance gains and shows how, in a simple test case, the use of connection pooling improves performance by a factor of three.[14]

The acceptance of the connection pool mechanism was affected by its proprietary nature. Because the WebSphere Application Server connection pool interfaces were not part of the Java standards (and therefore, not portable), many application developers chose not to use them. To maintain portability, some wrote their own connection pooling components, which were often fraught with problems due to poor implementation. Others did not use connection pools and paid the performance penalty for creating connections at runtime. Consequently, early performance crises were frequently caused by database connection issues.

Table 3 Thread-pool evolution

WebSphere Application Server Version 3.5	WebSphere Application Server Version 4.0	WebSphere Application Server Version 5.02	Future
Web container thread pool bounded, ORB thread pool unbounded	Bounded and unbounded configuration options for both pools	Runtime advice for tuning and adjusting thread pool sizes	Adaptive and autonomic thread pools

Even developers who used the WebSphere Application Server connection pool experienced difficulties, with the most common problem being the failure to return the connection to the free pool (which still happens today). When applications fail to release connections after use, the finite nature of connection pools surfaces. The connection pool runs out of available connections, applications wait longer to acquire a connection, and performance deteriorates. The performance and scalability benefits of making finite resources appear unlimited materialize only if the applications are implemented to make proper use of these services.

WebSphere Application Server Version 3.5 and Version 4.0. The need for connection pooling was quickly recognized by the Java community, and connection pooling was included in the JDBC 2.0 standard. WebSphere Application Server designers anticipated support of the JDBC 2.0 standard by introducing the JDBC 2.0 API for connection pooling in WebSphere Application Server Version 3.5, and JDBC 2.0-compliant connection pooling in WebSphere Application Server Version 4.0. The database connection pooling services are used within WebSphere Application Server by the EJB container (for CMP persistence) as well as by application developers writing direct JDBC calls from servlets or bean-managed persistence (BMP) beans.[2]

Although a standard connection pooling service was added to WebSphere Application Server for database access, many performance issues remained for applications accessing other back-end resources, such as IBM Customer Information Control System (CICS*)[15] or IBM Information Management System (IMS*).[16] Access to these back-end resources could also benefit from connection pooling. With no standard in effect, WebSphere Application Server designers again stepped in with proprietary support based on the VisualAge* Common Connector Framework (CCF).[17] CCF provided common services for accessing many back-end resources, including CICS, IMS, and IBM WebSphere MQ,[18] greatly simplifying development of these applications. CCF made these finite resources appear unlimited to the application developer. Similar to the evolution of JDBC, the need to standardize back-end access within J2EE was required to solidify usage. CCF evolved to become the J2EE Connector Architecture (J2CA)[2] standard, discussed next.

WebSphere Application Server Version 5.0. J2CA supports access from Java programs to a broad range of back-end resources in the enterprise. A *resource adapter* is a system-level software driver that plugs into an application server and enables a Java application to connect to various back-end resources. The specification provides the necessary interfaces for supporting these back-end resources as well as the interfaces for use by application programmers. Similar to the JDBC support, WebSphere Application Server uses the resource adapters both for EJB access and for direct application use of the available connection pool interfaces.

Thread pools. Applications accessing back-end connections acquire threads either from the Web container thread pool or the ORB thread pool. These pools contain the actual resources that execute a user request. Therefore managing the thread pools (to enable work to get done and to keep the work flowing through the system) is critical to WebSphere Application Server performance and scalability. We describe now the evolution of these pools, which is summarized in Table 3, the design challenges involved, and directions for future work.

Web container thread pool. The Web container initially supported a fixed-size thread pool. As requests came in from the Web server, they were handled by Web container threads until the threads were exhausted, and additional requests were queued. The thread pool could grow and shrink within a specified, bounded range. Although the bounded pool size constrained the available resources, the restriction was transparent to the application. Even so, there were situations in which this design was problematic.

First, consider a fixed-size thread pool and a downstream resource that also has a fixed-size pool. It appears that if we make the size of the two pools the same, then there should be no resource contention. Unfortunately, this argument fails to consider that sometimes an application requires multiple threads from the same pool, which may lead to a deadlock.

Second, some WebSphere Application Server sites experience very bursty request patterns. This requires the ability to temporarily grow the thread pool in order to handle the bursts transparently to the application. However, even providing an unbounded thread pool also presents potential issues, as experienced just downstream in the ORB pool.

ORB thread pool. The ORB threads are used for EJB container work when EJB calls are made directly to WebSphere Application Server through IIOP. These requests may originate from Java clients or from another WebSphere Application Server system, but not from a local servlet engine, which uses the Web container pool to process EJB calls.

The original ORB thread pool design was based on a different model from the Web container thread pool. The ORB thread pool started out as an unbounded thread pool without the ability to fix the maximum pool size. The primary reason for this design was to avoid causing a potential deadlock due to circular EJB calls. Consider, for example, an EJB in container A that calls an EJB in container B, and assume that the EJB in container B makes a call back to container A. If there is no available thread in container A, a deadlock occurs. An unbounded ORB thread pool tries to avoid this problem by ensuring that a new thread in container A is always available.

Unfortunately, an unbounded thread pool does not treat finite resources as finite. This design is susceptible to resource exhaustion in certain circumstances, for example, when a deadlock occurs. In the Web-container thread-pool discussion, we mentioned the potential of deadlocks in fixed pools requiring multiple connections. Now consider the same scenario with an unbounded thread pool, an application requiring access to multiple back-end connections, and a fixed number of back-end connections. The pool connections can still consume all the back-end connections and be deadlocked, with no connection being able to get the second required connection. After the deadlock occurs, the unbounded pool continues to create new threads for all incoming work, leading to uncontrolled growth. This scenario can lead to resource exhaustion, and proved problematic to certain WebSphere Application Server Version 3.5 applications.

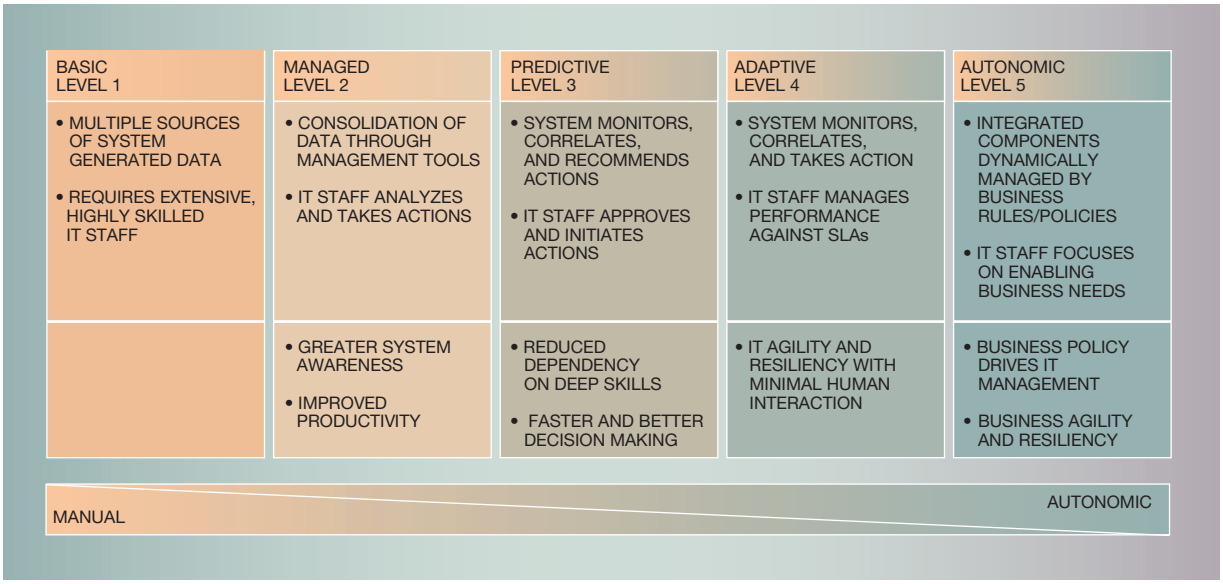
WebSphere Application Server Version 4.0. As the design of thread pools improved, more flexible behavior was built into both types of pools. In this release, the two designs were brought together, with common code for handling the pooling of threads in both the Web-container thread pool and the ORB thread pool. In addition, this design supported both bounded and unbounded configurations—the decision as to which configuration to use depended on the characteristics of the application.

WebSphere Application Server Version 5.02. Although WebSphere Application Server Version 5.0 thread pools self-adjust within the limits of their range, the settings must be configured by the administrator. The finite resources may appear unlimited to the application developer, but the responsibility for managing the resources is shared between the administrator and the WebSphere Application Server infrastructure. In this release an important step was taken toward removing the burden of resource management from the administrator, a goal of autonomic computing.[19] This release introduced the Runtime Performance Advisor,[1] which runs in the background, analyzing system resources and thread pool utilization and issuing tuning suggestions, in particular, suggestions for optimizing pool size.

Future directions in resource management. The autonomic levels discussed by Ganek[19] and shown in Figure 10 also apply to the evolution of WebSphere Application Server resource management. WebSphere Application Server Version 5.02 moved the product to the "predictive" stage in the autonomic evolution. This stage requires administrators to understand WebSphere Application Server and application resource management and to explicitly configure tuning parameters. Future autonomic enhancements are underway, into the "adaptive" stage, towards eventually attaining the autonomic goal of self-adjusting resources without any administrative interactions. At that stage, finite WebSphere Application Server resources appear unlimited to both the application and the administrator.

Memory. One of the primary motives for limiting thread-pool and connection-pool sizes is exercising control over memory resources. The creation of a thread or a connection requires allocation of memory. Memory for operating WebSphere Application

Figure 10 Evolving to autonomic operations



From *IBM Global Services and Autonomic Computing*, IBM White Paper, October 2002; see http://www-3.ibm.com/autonomic/pdfs/wp-igs-autonomic.pdf.

Server and executing applications comes from the JVM heap and is periodically reclaimed through garbage collection (GC). This mechanism, which removes memory management responsibilities from the application developer, is one of the strongest selling points of the Java environment. However, because GC significantly impacts performance, careful management of memory is critical to high performance and scalability. Specifically, in addition to controlling the creation of new threads, attention to overall object creation and usage is important. The more objects are created and destroyed, the more "garbage" is generated, and the longer and more frequently the garbage collector runs. GC represents overhead because when the garbage collector runs the application is not servicing requests.

Fortunately, GC design has made great strides alongside the WebSphere platform. In early WebSphere Application Server releases heap sizes were small, with a recommended maximum of 128 MB. These smaller heaps required more frequent GC runs, but each GC run was short, minimizing the performance impact on in-flight requests. When an application had large memory requirements, noticeable pauses occurred during the GC cycles.

As GC algorithms matured, situations when GC runs single-threaded, preventing other work from running in the JVM, occurred less often. Java Developer Kit (JDK**) 1.3.1,[2] supported in WebSphere Application Server Version 4.0 and WebSphere Application Server Version 5.0, included parallel mark and parallel sweep techniques and also introduced the optional concurrent mark technique.[20] These GC improvements allowed WebSphere Application Server applications to run with larger heaps, and thus with fewer GC runs. With the combination of faster processors and better GC design, GC pauses have become rare. WebSphere Application Server is now frequently run with 1 GB heaps. The 1.4.1 JVM, supported by WebSphere Application Server 5.1, further improves performance with incremental compaction techniques.

Nevertheless, even as garbage collectors have evolved, careful consideration of memory usage within WebSphere Application Server and WebSphere Application Server applications is still required. WebSphere Application Server uses techniques such as object pooling to minimize memory use. Application developers are also encouraged to pay close attention to memory use, and to use ob-

ject pools when appropriate.[5] In WebSphere Application Server Enterprise Edition Version 5.0, the platform provides an object pooling service for use by application developers. This capability allows developers to easily achieve the benefits of object reuse for complex and frequently instantiated objects.

Although the WebSphere Application Server platform is optimized for efficient resource utilization, high performance and scalability are achieved by combining it with applications that properly leverage its capabilities. In the next section, we look at the importance of optimizing the application and examine the ways application design has evolved toward a more effective use of the WebSphere Application Server platform.

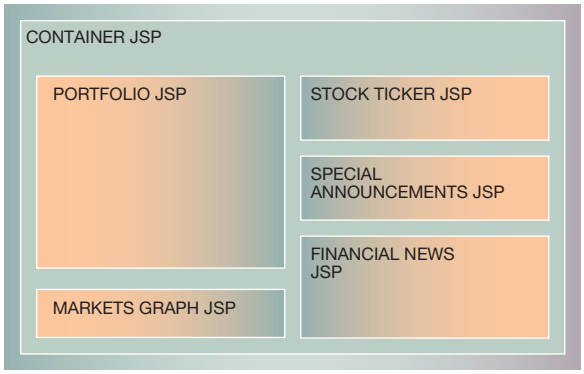
The evolution of caching and EJBs: Minimize interprocess calls

Whereas the evolution of WebSphere Application Server has incorporated an understanding of how users develop their applications, the applications themselves also evolved in order to take better advantage of WebSphere Application Server features. The HTTP session support has improved, and resource pooling has been adopted. The design of customer applications has changed to better take advantage of the trade-offs within J2EE. A mutually reinforcing cycle—the application server changes to meet application requirements, while applications change to take advantage of new application server features—has been a constant theme in WebSphere Application Server development over the past several years.

Often we find that the important lessons are the most painful ones. As children, we do not realize that fire can hurt until we get burned for the first time. In the same way, we have learned, rather painfully, that some of the ideas we had for developing WebSphere Application Server applications were less applicable than we thought. One of these lessons is that we need to make every call (especially extra process calls) count.

It is obvious, in retrospect, that invoking a local procedure is faster than a remote procedure call, regardless of the technology used (the fastest code is no code). It is also faster to access a cached, local copy of data than to retrieve data from a remote storage device. While this might belong in Computer Science 101, the appropriate application of these principles is far from intuitive, and it has taken us several tries to achieve the right balance of in-process and out-of-process calls in the WebSphere Application Server implementation. Understanding this balance has impacted both the WebSphere Application Server design and WebSphere Application Server application programming. For our first application of this principle, we examine content caching in WebSphere Application Server and how it has changed over time.

Figure 11 Typical screen layout



Content caching. The MVC architecture contains an implicit assumption that has proven to be its Achilles' heel in some cases, namely that every request for dynamic data from a client browser needs to proceed through each of the three layers (controller, model, and view) every time, regardless of the result ultimately presented to the user. Compare this dynamic approach with the handling of static Web content. The design of modern Web servers teaches that static Web content is read once from a fixed storage medium and cached in memory.[21] Although the dynamic nature of data in Web applications initially led us to believe that the presentation data needed to be generated every time, we learned that caching can also improve performance for slowly changing data.

Dynamic page generation and caching. We noticed that many of the screen designs being developed for the financial-services industry have a common layout, as illustrated in Figure 11. The page presented to the user has a number of distinct areas, each area presenting one kind of information. For a private investor, for example, the areas on the page could include a portfolio summary, a stock ticker that is updated every few seconds to show the latest stock prices, a graph of the market averages for the day,

Table 4 Evolution of content-caching capabilities

WebSphere Application Server Version 3.0	WebSphere Application Server Version 3.5.3	WebSphere Application Server Version 4.0.3	WebSphere Application Server Version 5.0	Future
Static pages	Servlet/JSP fragments	Cache replication	Command caching WebSphere Application Server plug-in or edge caching Web services server cache	Web services client cache

headlines of financial news, and special announcements from the financial-services firm. Most of the information on this page stays the same (is "read-only"). The low read/write ratio in this type of application is key to the performance optimizations that follow.

Each region on the page is represented by its own JSP and a corresponding fragment of Hypertext Markup Language (HTML), and these are all included in a global JSP (using the standard JSP include mechanism). This containment scheme, which is found in other applications as well, is the basis of Web portals.

Although all the regions contain dynamic data (not static data from an HTML file), data in some of the sections change at a faster rate. For instance, the stock ticker and the portfolio summary data may change in a synchronized fashion at fixed intervals; every 15 seconds, say, there may be a change in a stock price that causes some data to change. However, the special-announcement region may change much more slowly—perhaps only once a day or even less often than that.

We also realized that some of the sections are common to many users. For instance, the financial-news and the special-announcement regions might contain the same information for a large number of customers and would change for all customers at the same time.

When we understood how the data presented to the user change over time, it was a short step to realizing that caching the dynamically generated pages could provide applications with a significant performance benefit. Thus, one of the key performance features added to WebSphere Application Server was dynamic caching. We realized that caching could work at the model and view levels of the MVC architecture. The primary benefit of this caching was that it would enable us to avoid expensive and unnecessary process hops. If the servlet or JSP processing would not require a call to a database server or second application tier, the performance would improve. The evolution of dynamic caching is shown in Table 4.

WebSphere Application Server Version 3.5.3. The first part of the dynamic-caching implementation to be exposed allowed a WebSphere Application Server system administrator to identify a set of JSP pages whose output can be cached, and then to specify (1) the conditions under which the page can be cached, and (2) the conditions under which the page can be refreshed. The *cache key* combined information from the HTTP request and the HTTP session data that uniquely identified a page. The information on pages, their keys, and their refresh conditions was held in an XML (eXtensible Markup Language)[22] deployment file that allowed the EAR (enterprise application archive)[2] administrator to tune the cache (adding or removing pages, or adjusting the caching parameters) without affecting the code of the Web application at all.

Dynamic caching, which was introduced in the WebSphere Application Server Version 3.5.3 fix pack, has been a significant contributor to WebSphere Application Server performance and a performance differentiator for the product.[11]

WebSphere Application Server Version 4.0.3. As it turned out, it was not enough simply to identify what pages to cache and to specify under what conditions to cache them. Because WebSphere Application Server is usually deployed in a clustered environment, if a page is placed in a cache on one server, and later that page is invalidated on another server, then two identical requests from the same user could produce different results depending upon which server processed the request. To prevent this from happening, the dynamic-caching facility allows for replication of caches across cluster nodes—as

changes occur within one node they are propagated to the different members of the cache synchronization group.

Even though dynamic replication was part of the initial design of dynamic caching, its implementation evolved over time. The original plan was to include an RMI-based (remote method invocation-based) replication scheme, which was never deployed.[23] Early versions of WebSphere Application Server had featured a similar RMI-based HTTP session replication mechanism, which had gained a reputation for underperforming, so we decided not to ship the RMI-based replication with the first version of dynamic caching. However, soon afterwards, we became aware of a JMS-based multicast model that was used in a prototype application developed for the Web site for the 2000 Olympics. We adopted that approach and released JMS-based replication in WebSphere Application Server Version 4.03.

How does this relate to reducing the number of out-of-process calls? Simply put, it reduces the number in the most direct way, by avoiding calls altogether. A fully populated cached page for a JSP or a servlet Universal Resource Identifier (URI) not only avoids the servlet or JSP call, but all the downstream calls as well. Here we have a multiplier effect at work, where a hit to an upstream cache can avoid several downstream calls.

In the field, we have seen dynamic caching provide large, sometimes order-of-magnitude improvements in the performance of applications suited for this kind of optimization. Cox and Martin[24] document performance benefits up to a multiplier of 10 from applying dynamic caching after the fact to an existing RSS (RDF [Resource Description Format] site summary) servlet. We should point out that the experiment involved a simple servlet, and this order-of-magnitude improvement may not be reflective of a more complex application mix.

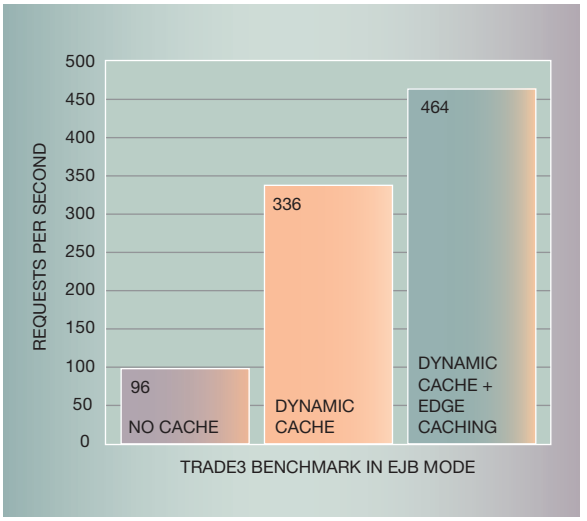
WebSphere Application Server Version 5.0 command caching. As successful as WebSphere Application Server 4.0 dynamic caching has been, it was not effective in all circumstances. In our example of a financial-services Web site, we described how the financial-news and special-announcement data could be held in the dynamic page cache for all users. We could hold the portfolio summary pages in cache for each user for a limited time, specified by a time-out parameter; if stock prices are updated every 15 seconds, then we could hold the page in memory for up to 15 seconds. However, we could do even more. Because the stock prices are updated every 15 seconds, any stock price remains the same for that period regardless of who makes the request. We cannot use dynamic page caching in this instance because each user's list of equities is different (that is, I care about stocks A, B, and C, while you care about X, Y, and Z). This realization that many EJB calls are for the same information—even when the page displayed is different—led to the introduction of the innovative WebSphere Application Server *command caching*, which is an implementation of the Command design pattern.[25] A command encapsulates an action as an object. The intent of the Command pattern can be stated as "encapsulate a request in an object, thereby parameterizing clients with different requests, queue or log requests, and support 'undoable' operations (that is, operations that can be backed out)."

Command caching is designed to provide benefits by allowing developers to create their own Command classes that can be either executed or undone, and that can execute either locally or remotely (behind an EJB session facade[26]). However, the most innovative feature of command caching is that it can significantly reduce the number of remote calls by caching the results of a remote EJB call within the command implementation.

Just as the dynamic-caching page-cache feature allows an HTML page to remain fresh for a period of time, or until it is explicitly marked "dirty" by an API call, a cached command can also be held for a period of time or until it is explicitly cleared. Command caching allows a system to take advantage of the fact that a single page may expand into many calls into the domain model. Many of those calls may have the same value over some period of time, while others are more time-sensitive.

For instance, consider the page for an item posted on an auction Web site. Many fields of the page (the item being auctioned, its description, its original bid price) remain the same until they are explicitly changed by the seller. Others, the maximum bid and the time remaining on the auction, are time-sensitive and change rapidly. Whereas command caching allows the same code to execute every time within the servlet code that populates the page, the commands (for example, GetTimeInsensitiveItemDetails, GetTimeSensitiveItemDetails) are treated differently; the former would be cached, whereas the latter would not.

Figure 12 Performance benefits of edge caching



Although command caching was considered for the original WebSphere Application Server Version 3.5.3 version of dynamic caching, it required a programming model change, and this led to some resistance to its introduction. Following the general acceptance of the Command pattern in J2EE programming, command caching was finally deployed as part of the programming model in WebSphere Application Server Version 5.0. Just as with page caching, the primary advantage of command caching is avoiding unnecessary computation, and more importantly, unnecessary process hops. By caching command results in the domain model, we can avoid repeated SQL calls, EJB calls, and other back-end invocations.

Future directions in content caching. As dynamic caching evolved, we discovered other places in which it is useful. For instance, in Web services, WebSphere Application Server Version 5.0 can dynamically cache Simple Object Access Protocol (SOAP) responses (with the cache key including elements of the SOAP request) on the server side in the same way that HTML page results are cached. We are currently implementing a Web services client-side cache that works at the same layer as command caching, allowing us in certain cases to avoid altogether sending SOAP messages over HTTP.

A most interesting development is the extension upstream of command caching, into the WebSphere Application Server plug-in and the edge server components. For instance, we now support caching HTML results from JSP pages in HTTP servers and in caching proxy servers. The bar graph in Figure 12[12] shows the result of running Trade3, the WebSphere Performance Benchmark Sample,[27] in EJB mode in three different configurations. The first bar represents the default configuration, with no dynamic caching. The second bar shows a 3× (factor of three) performance improvement in throughput through the use of dynamic and command caching. The third bar shows an additional 38 percent improvement gained by adding caching capabilities in the WebSphere Application Server plug-in.

Fowler's first law and local EJBs. The history of distributed computing has been one of a few small triumphs in an endless sea of failed predictions, lowered expectations, and general disappointments. The idea of making domain logic available over a network to any application that requests it is so seductive and so right that otherwise pragmatic and hard-boiled computer scientists have been falling over themselves for nearly 30 years in order to make that dream a reality. But rather than dwell on the disappointments in the history of distributed computing environments (DCE),[28] Common Object Request Broker Architecture (CORBA**)[7] and Distributed Common Object Model (DCOM),[29] we start with the version of this story that is most familiar to us—the promise and reality of distributed computing using EJBs.

When the EJB specification was first written, it was a visionary merging of a number of technologies and ideas. By building on the solid foundation laid in CORBA, and by adding new features, it was hoped that the success that (for the most part) had eluded CORBA would be achieved. CORBA was criticized for its complicated programming model, which was difficult to master. By grafting the simplified programming model from Java RMI onto the CORBA framework, it was hoped that the adoption of distributed computing by Java programmers would be hastened. However, that was not nearly all that went into even the first EJB specification. The specification also included a sophisticated transaction management mechanism, and even a mechanism for object persistence. Because these were seen to be critical requirements for developing complex business applications, its designers pinned their hopes on this combination.

However, there were some assumptions hidden in the very base of the EJB specification that would cause grief to many of its early adopters. From the beginning, it was assumed that EJBs would be shared across applications. That was, after all, the reason for distributed programming: the assumption that it would be cheaper and more efficient to invoke business-processing logic on a remote shared server than to invoke it local to the UI (user interface) logic. The second assumption was that any overhead would be small compared with the processing requirements; that is, the bulk of the time spent in executing a remote function would be in the business processing rather than in the delays of transferring parameter values and results over the network.

There were many grand designs that, in effect, decreed "the more tiers the merrier." In the accepted wisdom of the first J2EE specification, the typical topology of a distributed system would resemble the upper part of Figure 1, in which Web containers and EJB containers reside on separate WebSphere Application Server instances.

Soon after the first large-scale EJB applications moved into production, it was discovered that the assumptions did not play out as well as had been hoped. It was realized that while some EJBs are reusable across applications, far fewer have turned out to be reusable than first thought. A more troubling aspect—even when EJBs proved to be reusable—was that deploying reusable EJBs remotely from application-specific EJBs can create serious version-management issues.[30]

However, EJBs were not a failure by any stretch of the imagination. Other properties of EJBs—the ability to manage two-phase commit transactions declaratively, the ability to define object security declaratively, and the ability to provide standard object persistence—have been quite successful.

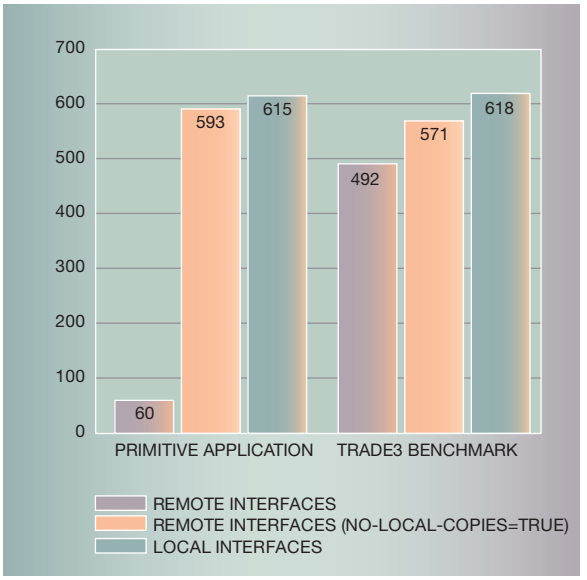
Java developers have become quite comfortable using EJBs, even though they have found that most deployed applications have a simpler topology than that shown in the upper part of Figure 1. In particular, developers realized that the same information is needed on both sides of the distribution boundary when using EJBs; thus, it is easier to have a single transformation (Java-to-HTML) when transferring results to the client than two (Java-to-Java and then Java-to-HTML). This reasoning led to a simpler topology in which Web containers and EJB containers are collocated, as shown in the lower part of Figure 1. In this arrangement, the Web container and the EJB container are deployed within the same WebSphere Application Server instance. This topology allows for linear scalability, even without the use of a second EJB tier, by using load balancing at the WebSphere Application Server plug-in layer. The performance advantage of this approach is that remote calls are eliminated entirely. The WebSphere Application Server ORB can detect when an EJB client and server are physically collocated within a JVM; when this is detected, the ORB "short-circuits" the calls by avoiding the use of socket connections between the client and server, and instead replacing the socket communication with Java method invocations.

WebSphere Application Server Version 3.5 and Version 4.0. Application developers discovered that an additional performance advantage of this topology was realized (without change to the application code) when using pass-by-reference, a feature referred to as *no-local-copies*. In short, the no-local-copies optimization turns off the serialization and deserialization of EJB parameters and return values, which means that both client and server code use the same set of value objects. In early WebSphere Application Server Version 3.5 performance tests, this optimization was found to improve performance in many applications with collocated Web and EJB containers by 10 percent or more.[31]

Soon this arrangement of collocating Web and EJB containers combined with the use of the no-local-copies optimization became the recommended topology for most EJB systems. In fact, the logic of this arrangement was even codified by Fowler[32] into what he refers to as "Fowler's first law of distributed object design," which simply states, "Don't distribute your objects."

WebSphere Application Server Version 5.0. So widespread and successful was Fowler's first law of distributed object design that it even made its way into the EJB 2.0 specification. In the initial versions of the EJB specification, every EJB object was a remote object; there was no other way of specifying EJBs. However, in the aftermath of the furor over the EJB 2.0 final specification 1 (which contained a version of CMP that was shown to be unworkable), an older proposal from the IBM team was "dusted off" by the EJB specification team. The proposal suggested the introduction of local entity bean objects, which operate solely within a single JVM and do not present a remote interface, as a solution to a number of problems inherent in trying to manage distributed cach-

Figure 13 Performance advantage of local interfaces



ing and prefetching of database results. This proposal was extended from entity beans to all EJBs, and was introduced into EJB 2.0 final specification 2, which is the version implemented by WebSphere Application Server Version 5.0.

We show the effect of these optimizations, and the additional benefits gained by local interfaces in Figure 13.[12] This chart shows the performance gain for both a simple test case (with minimal business logic) and a more representative (albeit still simple) application, Trade3. Three configurations were run: (1) remote interfaces only, (2) remote interfaces but with no-local-copies set to TRUE, and (3) local interfaces only. As the bar graph shows, a dramatic improvement is realized for the primitive application, and a modest 16 percent improvement for the Trade3 application. For the third configuration the implementation was rewritten to explicitly use EJB 2.0 local interfaces. As shown in Figure 13, additional performance benefits, above and beyond the no-local-copies case, are realized by the explicit use of the new API. The same conclusion, that using local interfaces to avoid remote calls results in significant improvements in overall application performance, was also reached by Cecchet, whose analysis of the JONAS and JBoss servers indicates that the results are not limited to WebSphere Application Server.[33]

The performance differences between local and remote interfaces lead us to wonder why so much time and effort have been invested into distributed computing when the numbers do not prove its benefits. As with so much in computing, we believe the answer may be found in Moore's Law. If we look back, we find that processor speed and memory capacity (which are directly tied to the number of transistors on each chip) have increased at a much faster rate than network bandwidth. For instance, 10 years ago, a high-end desktop machine might have had a 90 MHz processor, 64 MB of main memory, and a 10 Mbps Ethernet local area network (LAN) connection. Today a similar machine has a 2 GHz processor, a gigabyte of memory, but only a 100 Mbps Ethernet LAN connection. Thus, although processor speed has increased 20 times and memory size has increased 16 times, the data transfer rate has only increased 10 times.

The underlying assumption of distributed processing is that the cost of processing is high compared to the cost of network overhead. So, while in the past it would have made more sense to do business processing remotely from UI (user interface) processing because the cost of processing was higher; now that processing costs are lower it makes more economic sense to simply move to a more powerful processor, or several processors within a single machine, than to rely on remote processing over the network.

Nonetheless, there are still cases where distributed processing with EJBs makes sense. In particular, there are often good reasons that the UI and business code cannot run in the same machine. For instance, with Java clients it may be advantageous to run the shared business logic behind a firewall. In this case, we prefer not to download the business logic to the client machine for security reasons, or because the shared business logic can take advantage of resource pooling of the type described in previous sections. Similarly, when working with applications that we do not control (such as in a Web services scenario), we may still want to distribute our objects.

Evolution of CMP EJB performance. One of the more significant success stories in WebSphere Application Server performance has been the dramatic improvement in the performance of CMP entity beans over time. In the early version of WebSphere Application Server (and indeed most J2EE servers), CMP entity beans had a somewhat deserved reputation as a performance sinkhole. Luckily for developers, this situation has improved significantly. A summary of

Table 5 Evolution of CMP EJB performance optimization

WebSphere Application Server Version 3.5	WebSphere Application Server Version 4.02	WebSphere Application Server Version 5.0	WebSphere Application Server Enterprise Edition Version 5.0
Read-only flag	Optimistic and pessimistic concurrency	Local EJBs, preload optimization, store avoidance determined by container	Application profiling

the major performance optimizations is shown in Table 5.

WebSphere Application Server Version 3.5. At the time of this release, the level of the EJB specification was EJB 1.0. CMP support in this level of the container was optional, and few vendors offered it. Container-managed entity beans in EJB 1.0 were defined as remote objects (as were all other EJB types in EJB 1.0), and their bean implementation classes defined a set of attributes that were characterized in the deployment descriptor as being "container-managed" (meaning they were persistent in a database). However, the persistence support defined in EJB 1.0 was primitive; it was not even up to the standards of most object-relational mapping tools available at the time. For instance, in the specification there was no way defined to represent relationships between entity beans.

Regardless of the state of the EJB specification, the WebSphere Application Server 3.5 CMP entity implementation went beyond the specification in a number of ways. It implemented a form of EJB relationships by allowing the addition of special "getter" methods (those EJB methods prefixed with "get" that are used to retrieve attribute state from an EJB) on an entity bean class. (These methods use foreign-key information, stored as a persistent field in the entity bean, as an argument to a special finder method on the EJB home of the related entities to return the related beans.) Also, through the addition of a read-only flag on CMP EJBs, it remedied what many perceived as a flaw in the EJB specification.

To understand why the read-only flag was necessary, we have to understand the life cycle of CMP entity beans as defined in the EJB specification. CMP entity beans, can be created through a "create" method on an EJB home, or they can be instantiated from information held in an underlying data source through the use of a "finder" method on the EJB home. In either case, the actions of creating or finding the EJBs are usually contained within an outer containing-EJB transaction scope defined by the session-bean method that initiated the action. This is where the problem arises. The EJB specification also states that at the end of an EJB transaction, the entity state must be updated in the database. This is a very cautious approach that assumes that the container itself does not track the changes that are made to the state of the entities as they change during the course of the transaction. So, in the simplest interpretation of the specification, all entities touched during a transaction, whether or not they are actually modified, are written back to the database, even though that involves overwriting the same data already present in the database.

To eliminate this unnecessary step in the case where an entity bean was read but not modified during a transaction, WebSphere Application Server Version 3.5 included a read-only flag that extended the information in the deployment descriptor and could be applied to methods that did not modify the state of the EJB. For instance, "getter" methods could be marked by the developer as being read-only. The container was then able to determine on a case-by-case basis whether each EJB touched in a transaction had only methods with the read-only flag set on them called during that transaction, or whether other methods, not having that flag set, were called. In the latter case, the entity bean was considered "dirty" and written back to the database. In the former case, no database state was recorded, avoiding a number of unnecessary updates. This is consistent with the major theme of this section: avoiding process hops. By avoiding these unnecessary back-end SQL calls, we can substantially improve the performance of the application. This concept is referred to as *store avoidance*.

WebSphere Application Server Version 4.02. The next significant enhancement of EJB performance came

in WebSphere Application Server Version 4.02, which introduced a new way of treating the locking of database rows used by CMP entity EJBs. In this release, a new IBM-initiated deployment descriptor extension was introduced—a setting for "concurrency control" that had two values, "pessimistic" and "optimistic." When the choice was "optimistic" concurrency control, WebSphere Application Server used an overqualified update scheme to determine whether any columns in the target row changed during the transaction. WebSphere Application Server used all columns defined in the EJB deployment descriptor as CMP fields, with the exception of those columns having an ineligible column type (BLOB, CLOB, LONG VARCHAR, and VARCHAR having a length greater than 255). The addition of an optimistic concurrency scheme to the original pessimistic locking scheme (which used a "find by update" SQL statement) allowed certain types of transactions that have a low occurrence of collisions to operate concurrently without the chance of lost updates.

WebSphere Application Server Version 5.0. The most significant performance improvements to CMP support came with the introduction of WebSphere Application Server Version 5.0, which introduced support for the EJB 2.0 specification. As described earlier, one of the major innovations introduced in the EJB 2.0 specification was local EJB interfaces, which improved the performance of entity beans in the same way it improved the performance of session beans. However, that was not the only new item in the EJB 2.0 specification. In addition to local EJBs, the EJB 2.0 specification introduced EJB relationships (an IBM-initiated extension in WebSphere Application Server Version 3.5) as a supported feature. Also, in the WebSphere Application Server EJB support, store-avoidance logic was added directly to the EJB container, no longer requiring the explicit setting of the read-only flag.

The WebSphere Application Server implementation went beyond the specification to provide a significant performance gain to users of EJB relationships in CMP EJBs. It achieved this through the introduction of the *preload*, or "read-ahead" optimization. The idea behind EJB read ahead is so simple and straightforward that it comes as a shock to many that it is not a standard part of all EJB implementations. To elaborate, consider a situation in which we have an order CMP EJB that has a 1-to-many relationship to a number of OrderLineItem CMP EJBs. It is highly likely that whenever an order is handled, a query and an update to the line items within the order will be issued.

In a "default" implementation of EJB relationships, an SQL SELECT statement fetches the attributes of the order, and an additional SQL SELECT retrieves the OrderLineItem attributes. The preload optimization allows for the EJB relationships to be prefetched into a row cache, so the two SQL SELECT statements are combined at the outset into a single SQL JOIN. Then, when the CMP EJBs are "hydrated" or brought into memory (which only happens when an EJB is referenced either through iterating through a collection returned from a finder method or from an EJB relationship method), the attributes of the related EJBs are fetched from the existing row cache, and the additional SQL SELECT statements are avoided. Avoiding these additional SQL calls and the process hops that they entail can significantly improve the overall performance of the application. Performance advantages of 8–16 percent for two simple test cases, achieved through the use of WebSphere Application Server read-ahead capabilities, are documented by Gunther.[34]

There have been a number of additional performance improvements built into the CMP EJB implementation of WebSphere Application Server Version 5.0, besides the preload optimization. The introduction of *access intents* in Version 5.0 was a significant improvement in the locking behavior of CMP entity EJBs. Access intent is a specification of how an EJB will be used; for instance, whether by an optimistic or pessimistic locking scheme. WebSphere Application Server Version 5.0 Enterprise Edition introduced the idea of application profiling, which allows the same EJB to use different access intents under different circumstances (in the core and network deployment editions of WebSphere Application Server, only one access intent can be set per bean). WebSphere Application Server Version 5.0 also introduced support for SQLJ (Structured Query Language for Java) in the CMP implementation, which can improve the performance of CMPs by avoiding the dynamic compilation of SQL statements.

Future direction in EJB optimizations. The WebSphere Application Server implementation will continue to improve the performance of entity CMP beans over time. In WebSphere Application Server 5.02, we introduced the capability of batch updates for CMP changes. This takes advantage of the JDBC Batch Update feature in order to postpone updates until the very end of the transaction. Also in Web-

Sphere Application Server 5.02 we modified our access intent policies to provide better performance in the simplest cases by allowing for a default access intent "per bean," instead of "per method," as is now common. Although these small performance increases help, we see two larger performance enhancements on the horizon. First, we will support partial hydration of entity beans. This means that we will be able to selectively bring in the most commonly used parts of an entity bean when it is first requested, and then defer bringing in the rest of the bean until those attributes are referenced. This can substantially improve the performance of certain common use cases involving populating lists, which are now the exclusive domain of the "fast lane reader" pattern described by Marinescu.[35]

Also, we are examining ways of applying dynamic-caching technology (data-replication asynchronous-update mechanism, in particular) to CMP entity beans set up using Option A EJB caching. Currently Option A caching (which is defined by the EJB specification as bringing in an EJB once from the database and then holding the data values in memory indefinitely) is not compatible with WebSphere Application Server clustering because changes to a bean in one server are not propagated to other servers. Adding an asynchronous update mechanism to propagate those updates will allow Option A caching to be used in many circumstances and will significantly improve the performance of entity beans when it is used. This is perhaps the best example of avoiding process hops that we can envision. By restricting our EJB-based database activity to updates only, and not queries (as we do for HttpSessions), we should be able to substantially improve the performance of many EJB applications.

Conclusion.

WebSphere Application Server performance and scalability have improved with each release. The application requirements have driven WebSphere Application Server capabilities and, in turn, WebSphere Application Server capabilities have driven application design and deployment decisions. The performance and scalability improvements are based on the application of three major design principles: (1) optimize the design assuming failures will not occur, (2) make finite resources appear infinite, and (3) minimize the number of interprocess calls. A walk through WebSphere Application Server performance history clearly shows how understanding and applying these design principles enabled us to improve the performance of WebSphere Application Server and to support the development of highly performing Web applications.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc. or the Object Management Group, Inc.

Cited references and notes.

1. *WebSphere Application Server Version 5.1 Information Center*, IBM Corporation, http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp.
2. *Java 2 Platform, Enterprise Edition (J2EE)*, Sun Microsystems, Inc., http://java.sun.com/j2ee/.
3. *BEA WebLogic Server*, BEA Systems, Inc., http://www.bea.com.
4. *Oracle Application Server*, Oracle Corporation, http://www.oracle.com.
5. S. Joines, R. Willenborg, K. Hygh, *Performance Analysis for Java Web Sites*, Addison-Wesley, Boston (September 2002).
6. C. Sadtler, *WebSphere Application Server V5 Architecture*, Publication Number REDP-3721-00, IBM Corporation (2003), http://www.redbooks.ibm.com/redpapers/pdfs/redp3721.pdf.
7. *CORBA/IIOP Specification*, Object Management Group (OMG) (2003), http://www.omg.org/technology/documents/formal/corba_iiop.htm.
8. ECPerf Expert Group, http://www.theserverside.com/ecperf (free registration).
9. See results SPEC jAppServer2001 and SPEC jAppServer2002 published by the Standard Performance Evaluation Corporation (SPEC), www.spec.org.
10. For example, in the SPEC jAppServer results, balancing the workload was done using round robin DNS (domain name service) (BEA WebLogic Server 8.1 or Oracle Application Server 10g).
11. T. Dyck, "Web Server Brains," *PC Magazine* (May 22, 2001).
12. R. E. Willenborg, "Optimizing Performance with WebSphere V5.0," presented at *IBM developerWorks 2003 Technical Conference,* New Orleans (2003) (available from the author).
13. R. E. Willenborg, "WebSphere 5.1 Performance Update," presented at *IBM eBusiness University,* Las Vegas (January 2004) (available from the author).
14. H. Gunther, "WebSphere Performance Best Practices for Performance and Scalability," White paper, IBM Corporation (2000), available at: http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf.
15. *Customer Information Control System (CICS)*, IBM Corporation, http://www.ibm.com/software/htp/cics/.
16. *Information Management System (IMS)*, IBM Corporation, http://www.ibm.com/software/data/ims/.
17. D. H. Steinberg, *Using J2EE Connectors: Simplifying the Connections between Application Servers and Enterprise Information Systems*, IBM Corporation (August 2001), http://www.ibm.com/developerworks/ibm/library/i-j2eeconn/?dwzone=ibm.
18. *IBM WebSphere MQ*, IBM Corporation, http://www.ibm.com/software/integration/wmq/.
19. A. G. Ganek and T. A. Corbi, "The Dawning Of The Autonomic Computing Era," *IBM Systems Journal* **42,** No 1, 5–18 (2003).
20. S. Borman, *Sensible Sanitation—Understanding the IBM Java Garbage Collector, Part 2: Garbage Collection*, developer-

Works, IBM Corporation (2002), http://www.ibm.com/developerworks/ibm/library/i-garbage2/.
21. D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern Oriented Software Architecture, Volume 2*, John Wiley & Sons, Inc., New York (2002).
22. *Extensible Markup Language (XML) 1.0*, World Wide Web Consortium (W3C) (2000), http://www.w3.org/TR/REC-xml.
23. *Java 2 Platform, Standard Edition (J2SE)*, Sun Microsystems, Inc., http://java.sun.com/j2se/.
24. J. S. Cox and B. K. Martin, "Exploiting Dynamic Caching in WAS 5.0, Part 1," *e-Pro Magazine* (July/August 2003), http://www.e-promag.com/.
25. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, Reading, MA (1994).
26. D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies, 2nd Edition*, Prentice Hall, New York (2003). See Core J2EE Patterns at http://www.corej2eepatterns.com/Patterns2ndEd/SessionFacade.htm.
27. *Trade3: WebSphere Benchmark Sample*, IBM Corporation, is available for download at http://www.ibm.com/software/webservers/appserv/benchmark3.html. The package includes application code, primitives, design documentation, and source code.
28. *OSF Distributed Computing Environment (DCE)*, The Open Group, http://www.opengroup.org/dce/.
29. *Distributed Component Object Model (DCOM)*, Microsoft Corporation, http://www.microsoft.com/com/tech/DCOM.asp.
30. K. Brown and K. Botzum, *Deploying Multiple Applications in J2EE 1.2*, developerWorks, IBM Corporation (2003), http://www.ibm.com/developerworks/java/library/j-deploy/.
31. *WebSphere Application Server V3 Performance Tuning Guide*, Publication Number SG24-5657-00, IBM Corporation (2000). Available at http://www-900.ibm.com/cn/support/library/sw/download/sg245657.pdf.
32. M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, Reading, MA (2002).
33. E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and Scalability of EJB Applications," *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, Seattle, WA, ACM, New York (2002).
34. H. Gunther, "Optimizing WebSphere 5.0 Performance Using EJB 2.0 Caching and Read-Ahead Hints," *WebSphere Journal*, SYS-CON Media, Inc. (March 2003), http://WebSphereDevelopersJournal.com.
35. F. Marinescu, *EJB Design Patterns: Advanced Patterns, Processes and Idioms*, John Wiley & Sons, New York (2002).

Accepted for publication January 16, 2004.

Ruth Willenborg *IBM Software Group, PO Box 12195, 3039 Cornwallis Road RTP, NC 27709 (rewillen@us.ibm.com).* Ms. Willenborg is a Senior Manager in the WebSphere development organization where she is responsible for WebSphere Application Server performance. She has 17 years of experience in software development and is co-author of *Performance Analysis for Java Web Sites* (Addison-Wesley, 2002). She holds a B.S. in computer science from Duke University.

Kyle Brown *IBM Software Group, PO Box 12195, 3039 Cornwallis Road RTP, NC 27709 (brownkyl@us.ibm.com).* Mr. Brown is a Senior Technical Staff Member with the WebSphere Services group, where he advises IBM's largest WebSphere clients in the application of design patterns and best practices for Java, J2EE, and Web services. He is the lead author of *Enterprise Java Programming with IBM WebSphere, 2nd Edition* (Prentice-Hall, 2003) and has written or contributed to six other books and over forty papers on Smalltalk, Java, OO Design, Patterns, J2EE, and WebSphere. He has a B.S. degree in computer engineering from the University of Illinois at Urbana-Champaign, and an M.S. degree in computer engineering from North Carolina State University.

Gennaro (Jerry) Cuomo *IBM Software Group, PO Box 12195, 3039 Cornwallis Road RTP, NC 27709 (gcuomo@us.ibm.com).* Mr. Cuomo has been a core member of the WebSphere development and architecture team since its inception in 1998. He is a Distinguished Engineer and currently the CTO of the WebSphere Technology Institute, which is responsible for developing technologies for the next generation of WebSphere products.