J2EE and .NET (RELOADED)
Yet Another Performance Case Study

Based on
The Middleware Company Application Server Baseline Specification
http://www.middleware-company.com/casestudy/

The Middleware Company Case Study Team
June 2003

The Middleware Company
Performance Case Study

casestudy @middleware-company.com

www.middleware-company.com

Table of Contents

1 Summary

This report contains the results of a performance case study conducted by The Middleware Company, with the help of an Expert Group, and J2EE and .NET platform technology vendors.

The case study is based on a specification created by an invited panel of J2EE and .NET industry experts and practitioners, as described in Section 2 (Prerequisite Reading).

In the J2EE arena, four of the leading application servers were initially tested on Linux and Windows. From these four, the two fastest were selected. These two then became the main focus of tuning and testing. This report contains the results of testing these two J2EE application servers with both a JSP/Servlet implementation and a JSP/EJB implementation of the specification, plus the Microsoft .NET implementation of the specification.

The case study was divided into three main testing areas:

- **A Web Application Test**. This tested performance when hosting a typical, web application with steadily increasing user loads. This test used two different databases, Oracle 9i and Microsoft SQL Server 2000.

  The test results showed that both .NET and the fastest J2EE platform performed approximately the same. The J2EE solution was slightly better than the .NET solution (about 2%) when using Oracle 9i. When using Microsoft SQL Server, the .NET solution was slightly better than J2EE (about 11%). In general the J2EE implementations performed equally well against both databases. The .NET implementation performed almost the same as J2EE when using Oracle 9i and slightly better when using Microsoft SQL Server.

- **24 Hour Reliability Test.** This tested the sustainable performance and reliability of platforms over a 24-hour period as it runs a transaction-heavy test script against the web application under a constant, peak-throughout, user load. For this test the database used for each codebase was the one it appeared to perform most reliably and conveniently with, For both the J2EE codebases the 24-Hour Reliability Test was run with Oracle 9i. For the .NET codebase the 24-Hour Reliability Test was run with Microsoft SQL Server.

  The results of this test were that the fastest J2EE and the .NET platform performed almost identically, with less than 2% difference in performance.

- **Web Services Test.** This tested the performance of the application server as it hosts a basic web service via SOAP 1.1.

  The results of this test showed that the .NET platform outperformed the fastest J2EE platform, by over 200%.

In general the J2EE EJB implementation outperformed the JSP/Servlet implementation. The fastest J2EE application server, App Server X, far outperformed the slower J2EE application server, App Server Y, in every category of testing. The rest of the report discusses in great detail how these results should or should not be interpreted, the generalizations that can or cannot be made, the experiences and difficulties we encountered during this case study, some interesting decisions we made (with discussion and speculation regarding the outcome had we made different choices), and our theories on why we saw the results we did.

Drawing less strident conclusions is always less troublesome. The study shows that none of the platforms tested should be disqualified simply on the basis of performance. They are all contenders. Statements such as "J2EE can never win on Windows because that's Microsoft's home turf and they run native" or "Microsoft can't do enterprise applications" can safely be dispelled as nonsense, especially, with regard to performance.

Section 5 (what is our agenda), Section 6 (what conclusions can you draw), Section 11 (funding and participation), Section 12 (other disclosures), Section 13 (anomalies) and other such parts of this report attempt to address some of the politics that such reports must inevitably tolerate.

Section 7 describes the spec categories addressed by this case study. Section 8 describes the codebases used in this case study. Section 9 describes the tests used in this case study. Section 14 describes the test lab (hardware and software).

Section 10 discusses the application servers used in the study and explains why the J2EE application servers could not be named.

Separate sections are dedicated to experiences and anecdotes in tuning each app server platform, each codebase, and each database.

And of course, detailed data and charts of results for the various tests are included.

2 Prerequisite Reading: The Specification Documents

This case study is based on The Middleware Company Application Server Baseline Specification, available at:

http://www.middleware-company.com/casestudy/

It is best to read or at least skim that specification prior to reading the results of this case study.

The specification answers questions such as:

- Why a functional and behavioral specification that can be implemented on diverse platforms?
- Why a specification based on PetStore, of all things?
- How is this different from SPECjAppServer and TPC-W?
- What aspects of this case study are dictated by that specification, and what political and technical constraints did that specification impose on this study?
- What tests should submissions expect?
- What hardware and software combinations are possible?
- What is the functionality of the test application?
- How the database is structured (what are all the tables and relationships)? What is the DDL?
- What data can be cached? Where? For how long? How much of each type of data can be cached? Are third party caching products allowed?
- What's on the web pages?
- How is user authentication and session management performed?
- And many more.

Other material at the website specified above includes:

- Frequently Asked Questions
- Interesting debates amongst the Expert Group members, and feedback currently not incorporated by the specification
- Summaries o f incorporated feedback from the October 2002 case study
- Codebases used in this case study
- Raw material useful for third parties to create and submit codebases for case studies
- Press Coverage

3 Overview

In October 2002, The Middleware Company published the results of a case study that compared the performance of J2EE vs. .NET.

The Middleware Company published its comparison of the performance and scalability of .NET and J2EE based on the familiar Pet Store application. The results of this case study created much controversy within the J2EE development community when they showed that .NET had outperformed J2EE.

Subsequently, The Middleware Company responed as follows:

- In the months after October 2002, we cataloged all the feedback, suggestions, and complaints from the study
- In February 2003, we invited notable industry experts to create that Expert Group to help us create a framework for such case studies in the future
- With the help of the Expert Group, authored The Middleware Company Application Server Baseline Specification, a functional and behavioral specification of an application that can be used in case studies (architectural, design, performance, productivity, interoperability, etc.) of applications running in diverse application server environments, including J2EE and .NET
- In April 2003, made the specification public and proactively invited other industry experts and journalists to review and comment
- Produced modified codebases with all the feedback incorporated, as starting points
- Documented the changes we made to the codebases and what effect those changes appeared to have had, in small performance tests
- Invited industry experts to review and audit the codebases
- Invited third parties to submit codebases that conform to the specification
- Requested third party submitters to document their submission (architecture diagrams, docs, etc.) and provide a list of changes from prior submissions

Some detail about these activities is found in the rest of this document, and much more detail about these and related activites can be found at The Middleware Company Case Study website specified above.

4 A “Case Study”? Not a “Benchmark”?

This effort is called a “case study” and not a “benchmark” for several reasons.

First, the term “benchmark” evokes emotional reactions from people. That is counterproductive.

Second, the term “benchmark” implies a more formal structure and process (often with committees and voting) than what we have. The process we employed was less formal, more collaborative and more open.

Third, the term "benchmark" implies that the only important things are the final results. This is far from the truth. We feel that the final results of a performance case study are an important data point, but not the most important aspect of the case study. The debates and discussion between the participants, Expert Group members, and the community, scrutiny of decisions made in the specifications and the codebases, documentation of experiences, difficulties, and interesting anecdotes from the case study, open discussions with participants, and more, are all more informative, educational, and important, than merely looking at the results.

Lastly, a performance case study is just one kind of case study made possible by The Middleware Company Application Server Baseline Specification.

5 Why are we doing this study? What is our "Agenda"?

We are compelled to write sections such as this one, due to the great amount of controversy we created after the October 2002 performance case study, where we "simply published results." We took a lot of heat because .NET outperformed J2EE in that configuration, and our community consisted mostly of Java/J2EE fans and practitioners.

We will start with what our agenda isn't. It is not our agenda to demonstrate that a particular company, product, technology, or approach is "better" than others.

Simple words such as "better" or "faster" are best used by simpletons. Life, especially when it involves critical enterprise applications, is more complicated. We are doing our best to openly discuss the meaning (or lack of meaning) of our results and are going to great lengths to point out the several cases in which the result cannot and should not be generalized.

Our agenda is to be a profitable consulting company.

We do not have reasons to disbelieve industry analysts' consensus estimates of technology market share: that in the medium to long term, approximately 40% of the app server market will be J2EE, 40% will be .NET, and 20% will be Other.

To help our clients in the future, we believe we need to be experienced in and be proficient in both J2EE and .NET. We are conducting serious experiments such as this one, because they are great learning experiences, provide very interesting work for our staff, and we feel that every professional firm should conduct some learning experiments to provide their clients with the best value.

And if we go one step further and ask technology vendors to sponsor the studies (with both expertise and expenses), we involve the community and known experts, and we document and disclose what we're doing, then we can:

- Lower our cost of doing these studies
- Do bigger studies
- Do more studies
- Make sure we don't do anything silly in these studies and reach the wrong conclusions
- Make the studies learning experiences for the entire community (not just us)

6 What conclusions can be drawn from these (or any) performance results?

For enterprise applications, performance is important. But not all-important.

We have found among our customers that during the selection of enterprise application server platforms, performance is simply not the top selection criterion -- and often not even among the top five.

Other considerations, such as the perception of the vendor as an "enterprise company", conformance to standards, strength of the ecosystem around the product, the degree to which the product is a defacto standard among its class, stability of the product or technology, ease of use, IDE integration and other tools support, total cost of ownership (TCO), portability and lock-in (or lack thereof), scalability, and many other such factors often outweigh Performance, among our customers.

Note: Performance is not the same as Scalability. We believe that Scalability is more important than Performance.

Furthermore, out of necessity, performance case studies such as this one choose a particular deployment configuration (hardware-software platform combination) or two. And they choose a particular application (test workload) or two to represent typical customer applications. Journalist Timothy Dyck of eWEEK commenting on our work in his article *Benchmarks Wanted,* states:

> "Every benchmark, no matter how well or how poorly designed, models a very particular usage pattern and workload. Organizations with similar workloads will find the benchmark details and results highly valuable; organizations using different application architectures will know how to read the results in context. In both cases, full disclosure is key."
>
> [http://www.eweek.com/article2/0,3959,1085334,00.asp]

So what use are performance results then? Please use your judgment, look at the details, and use them in context.

7 Categories addressed in this case study

The application described by The Middleware Company Application Server Baseline Specification can be implemented in any programming language, on any application server platform, using any database.

But the spec currently also defines three specific "categories" that describe certain types of applications more precisely.[1]

The three categories the spec currently defines are:

- J2EE-EJB-CMP
  (A J2EE application built using J2EE, Enterprise JavaBeans, specifically using CMP2 (Container Managed Persistence, v2)
- J2EE-SERVLET-JSP
  (A J2EE application built using the JSP and Servlet features J2EE, but explicitly not using EJBs)
- .NET-C#
  (A .NET application built according to Microsoft's best practices guidelines for the enterprise, significantly revised since the first tests by The Middleware Company in October 2002)
  This category was called DOT-NET-CLASSIC by the Expert Group who helped write The Middleware Company Application Server Baseline Specification, but we felt that calling the category .NET-C# in this document was more descriptive. We have not

[1] More categories will probably be defined in the future.

changed the definition of the category. We are simply using a name for the category that we feel is more descriptive.

The J2EE-EJB-CMP category thus represents the "full" classical J2EE blueprint. The .NET-C# category represents the Microsoft-recommended n-tier .NET blueprint as documented in their Prescriptive Architecture Guidance documentation on the Microsoft Developer Network Web site (MSDN).

Comparing codebases in the J2EE-EJB-CMP and J2EE-JSP-SERVLET category allows us to measure the effect of EJB and its various features, such as entity beans and CMP.

8 Codebases used in this case study

This case study used the following codebases for the categories described above.

8.1 Codebase for the .NET-C# category

To obtain a codebase for the .NET-C# category, The Middleware Company requested the creator of the .NET codebase in the October 2002 performance case study, Microsoft, to:

- Incorporate community feedback from the October 2002 case study
- Make the codebase compliant with The Middleware Company Application Server Baseline Specification
- Make the codebase architecturally parallel to the J2EE codebases
- Produce an architecture diagram for the .NET codebase
- Document a list of changes they made to the original codebase

Microsoft did this.

Like codebases for other categories, each version of the codebase they provided was posted to the Expert Group collaboration website. Expert Group members, and later, others were invited to scrutinize and code this codebase for compliance to the spec and for "benchmark specials." No member of the expert group raised objections to the Microsoft implementation in adherence to best practice enterprise design (for example, cleanly abstracted UI, Business and data tiers, proper exception handling, etc.).

The practices used were base on the Prescriptive Architecture Guidance documentation and enterprise patterns for .NET available at the .NET Architecture Center

http://msdn.microsoft.com/architecture/

A full description of the patterns and practices used in developing the Microsoft PetShop is available from:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/petshop3x.asp

8.2 Codebase for the J2EE-EJB-CMP2 category

A "starting point" codebase for this category was produced by The Middleware Company by incorporating all the feedback from the community, and from staff members, after the October 2002 case study.

The changes made were listed, and their approximate performance impact was informally measured in small performance tests. The performance impact was listed along with the description of the change.

Like codebases in other categories, each version of the codebase was posted to the Expert Group collaboration website, and scrutiny and feedback was solicited.

8.3 Codebase for the J2EE-SERVLET-JSP category

The codebase used for this category is the open source JPetStore, a J2EE implementation of PetStore that does not use EJBs.

This codebase was suggested as a candidate for J2EE performance case studies by many reviewers of the October 2002 case study.

We involved the lead author and project leader of JPetStore, Clinton Begin of ibatis.com (who was also an Expert Group member); in making sure that the appropriate modifications were made to JPetStore to make it compliant to the spec.

Like codebases in other categories, each version of the codebase was posted to the Expert Group collaboration website, and scrutiny and feedback was solicited.

9 Tests Used In This Performance Case Study

The performance case study consisted of three suites of distinct tests which provide a fairly comprehensive overview of the performance and scalability characteristics of the codebases in all categories. The three suites were:

1) Web Application Test
2) 24-Hour Reliability Test
3) Web Services Test

The tests measured throughput curves, recorded as passed transactions per second, which each transaction consisting of a single 'page' in the application (hence, readers can interpret 'throughput' or 'transactions per second' (tps) as 'web pages per second').

9.1 Web Application Test

This test exercises most of the basic functionality of the application. The tests were run to determine a throughput curve as well as measure response times for the application from low to high user loads, pushing each product to high levels of stress to determine the maximum number of users supported in an 8 CPU application server configurations.

This test contains a fairly even mix of "reads" and "writes." The test scripts simulated users navigating through various portions of the website (browsing and searching), authenticating themselves and signing in, adding items to the shopping cart, and 50% of shoppers purchasing (checking out). Explicitly closed as well as abandoned sessions were included in the test scripts, with 50% of the signed in users explicitly singing out.

The test scripts were executed with an average 5 second think-time (actual think time for an individual test client was a random number chosen between 50% and 150% of that value; in other words, between 2.5 seconds and 7.5 seconds).

In the October 2002 performance case study, the think time was 10 seconds, +/- 50%. Thus, this performance case study stressed the servers twice as much at the same user load as the October 2002 study did. This was done to reduce the total number of users required to saturate the application servers.

Page-level output caching was disabled such that the servers had to process each request received.

User loads were ramped up gradually. After every half hour, 500 users (clients) were added, and the system was allowed to run with the new load for the half hour. (Thus total execution time for each data point was one half hour).

The test was performed with **no image download**, to measure the characteristics of the base application server engine (as opposed to also measuring static HTML and image serving characteristics). This test indicates how well an application server product handles application logic, including server-side scripting, session management, object activation and polling, and database connectivity.

In contrast, the October 2002 performance case study conducted this test both with and without image download.

All application server codebases were run on an 8 CPU machines. In contrast, the October 2002 performance case study ran each codebase on 2, 4, and 8 CPU machines.

For each codebase, the Web Application Test was conducted against two different databases: Oracle 9i (v9.2), and Microsoft SQL Server 2000.

In contrast, the October 2002 performance case study ran each codebase only against one database (the database against which it performed best in preliminary tests).

9.2 24-Hour Reliability Test

This test was designed to measure the reliability and sustainability of the transaction processing throughput of each implementation. The test script consisted of users logging in, and then proceeding to individually order 100 items. For each item ordered, the checkout process was completed, with the last step in this process being the actual placement of the order that activates a transaction, followed by a logout at the end of the script. Each user therefore completes 100 individual transactions during a user session. The test was run at a user load providing peak throughput for each product for duration of 24 hours to show if this throughput was sustainable.

9.3 Web Services Test

This test measures the performance characteristics of Web Services for codebases in all categories. The test a configuration, was as follows: 100 distributed physical computers each simulated multiple users making direct SOAP calls to activate the Web Service. This test measures the ability of the application server to handle incoming SOAP requests and act as a Web Service provider. We call this method of measuring Web Services performance *Direct Activation of the Web Service*.

In contrast, the October 2002 performance case study, also measured *Indirect Activation of the Web Service*, via a proxy object. In this configuration, two physical application servers are configured, one to act as the remote Web Service provider (as in the *Direct* test), and the other to act as a Web Service client. The 100 physical client computers generating load make HTTP/HTML requests to the application server computer, which in turn makes SOAP requests the Web Service provider machine. This test is designed to measure the application servers Web Service client performance and performance when making remote SOAP-based object activations.

This case study did not include the *Indirect Activation* test, and focused instead on the Web Service hosting capacity of the platforms tested.

9.4 Diagram of Complete Test Matrix

The following diagram summarizes all the tests conducted in this performance case study:

Codebase	Web Application Test with Oracle 9i	Web Application Test with SQL Server 2000	24-Hour Reliability Test with best database for this codebase	Web Services Test with best database for this codebase
J2EE-SERVLET-JSP on J2EE App Server X	✓	✓	✓	✓
J2EE-EJB-CMP2 on J2EE App Server X	✓	✓	✓	✓
J2EE-SERVLET-JSP on J2EE App Server Y	✓	✓	✓	✓
J2EE-EJB-CMP2 on J2EE App Server Y	✓	✓	✓	✓
.NET-C# on Microsoft .NET	✓	✓	Tested on Microsoft SQL Server 2000 and Oracle 9i	Tested on Microsoft SQL Server 2000 and Oracle 9i

10 Application Servers Used In This Performance Case Study

10.1 The .NET-C# category

Since there are other implementations of portions of the .NET framework and platform, such as Mono, it is worth stating that for the .NET-C# category, we use Microsoft's .NET Framework v1.1, which is part of Windows Server 2003.

10.2 The J2EE-SERVLET-JSP and J2EE-EJB-CMP2 categories

For these categories we used two popular J2EE application servers, which will be named as J2EE Application Server X, and J2EE Application Server Y, to steer clear of End-User License Agreements (EULAs) that make it problematic to disclose the identities of application servers in conjunction with the publication of performance numbers.

Preliminary experimentation began with four J2EE app servers, W…Z, and X and Y were chosen because they exhibited the best performance among the four, in preliminary tests conducted after tuning had been performed by consultants knowledgeable in those app servers.

App Server X was clearly the best performer among the J2EE app servers during preliminary tests (in this configuration!! See Section 13), and therefore an obvious inclusion, even though the vendor declined to participate. (See Section 11 for more about invitation and acceptance).

App Server Y was the best among the rest, and the executives and engineers of App Server vendor Y, were enthusiastic and spirited about participating and were pleasant to work with.

11 Funding & Participation: Were The Application Server Vendors Invited?

Vendors of all application servers used in the case study were invited to participate in the performance case study.

Microsoft accepted our invitation, and needless to say, agreed to be named.

The Vendor for J2EE Application Server X declined our invitation to participate, and declined to be named.

The Vendor for J2EE Application Server Y accepted our invitation to participate, but declined to be named.

All vendors who accepted, were expected to make contributions to the project, in the form of making equipment, software, and lab space available to us, making senior technical resources (such as lead engineers and architects of the products, and engineers specializing in performance) available to us, helping us troubleshoot technical problems, and reimbursing expenses. The vendors who accepted our invitation did so.

The lab and machines were provided by Microsoft and were located in Redmond, WA. All hardware, software and related aspects of the lab were under the control of The Middleware Company.

All vendors who accepted were invited to visit the lab for any and all portions of the performance case study.

Secure access, over the web, to the machines in the lab, was also provided to, and used by, application server vendors who had accepted our invitation.

For application sever vendors who accepted, and who were not local to the Redmond area, The Middleware Company offered to reimburse the airfare, ground transportation, hotel and other travel costs of their representatives, from the project budget.

Members of the Expert Group that helped author the Specification were all also invited to be present in the performance lab and observe the tests as they were being conducted, but we did not offer to reimburse their travel expenses.

12 Other Disclosures

At the time of the publication of this report The Middleware Company is doing, has done, and/or may be doing business with any or all of the following companies mentioned in this report: Microsoft, Oracle, App Server Vendors W, X, Y, and Z, or with companies that those companies have acquired with the last eighteen months.

Moreover, The Middleware Company is an independently operating but wholly owned subsidiary of Precise Software Solutions (www.precise.com, NASDAQ: PRSE).

On its website, Precise states that Microsoft is a Strategic Partner. Precise has an OEM relationship with Microsoft, which has licensed Precise/StorageCentral SRM technology for use as an optional add-on with Microsoft's Windows-Powered Server Appliance Kit (SAK).

http://www.precise.com/Partners/Strategic/

Precise also cites several J2EE Application Server Platform Vendors as Technology Partners:

http://www.precise.com/Partners/Technology/

Among Precise's product offerings is InDepth for J2EE, which is a suite of performance monitoring and management tools for the J2EE application server market:

http://www.precise.com/Products/Indepth/J2EE/

Precise may have business relationships with Microsoft, Oracle, and App Server Vendors W, X, Y, and Z.

But wait, there's more. Precise is in the process of being acquired by Veritas (www.veritas.com, NASDAQ: VRTS). It is possible and likely that Veritas may have business relationships with Microsoft, Oracle, and App Server Vendors W, X, Y, and Z. Please visit the Veritas website for more information.

13 J2EE App Server X much better than J2EE App Server Y – An Anomaly?

While reading the results below, you will find that J2EE App Server X is faster than J2EE App Server Y.

Not just faster, but much faster.

Is this an anomaly? Did we miss something? Did we screw something up?

This measurement—its existence, its appearance, and its potential perception by our readers—caused us great concern during the case study. This section attempts to explain this apparent anomaly.

Recall that in our test configuration, the HTTP server, the Servlet/JSP container, and the EJB container (if applicable) were all run on the same physical machine. This configuration gave some profound advantages to certain app server architectures or deployment configurations, over others.

App Server X performed much better than App Server Y in our configuration because App Server X's integrated HTTP server was able to handle the high volume of client connections, whereas App Server Y's was not, and relied on a "connector" or "plug-in" to bridge from another commercial strength web server such as Apache or IIS to the Servlet/JSP/EJB container.

The problem, for those app servers with integrated HTTP servers that did not scale well, stemmed from (a) contention for the CPU by the web server, (b) inefficiencies caused by the marshalling and interprocess communication across the connector, and (c) inefficiencies caused by the impedance mismatch of multiple thread pools in different containers trying to cooperate with each other under high load.

Sadly, we found that the majority of the J2EE Application Servers suffered from including poorly performing integrated http servers.

Recall from Section 11 that application server vendors were invited, and that the vendor of Application Server Y (the slower one) was not only invited, but intimately involved during the case study, and in particular to solve the problem being discussed here.

In fact, among the reasons we had chosen App Server Y over other app servers (other than X) were that (a) this effect was even more pronounced in other app servers, and (b) the vendor of App Server Y was enthusiastic about working with us in fixing this problem.

We demonstrated this problem to the engineers from App Server vendor Y, by running various tests (as they watched), and also with profiling graphs and tables produced by Precise InDepth for J2EE (a performance analysis and profiling tool), and Mercury LoadRunner.

They acknowledged our diagnosis and the fact that the problem did exist. For approximately six continuous weeks, the vendor of App Server Y tried to fix this problem, by producing new patches, which gradually made the code surrounding this connector architecture more efficient.

During these weeks, 50+ emails were exchanged with engineers from App Server vendor Y every week, with phone/conference calls happening s everal times a week, and occasionally more than once per day.

During this process several other unrelated smaller performance issues were discovered and quickly fixed, and any techniques that were generic enough to be also applicable to App Server X, were applied.

Working with the vendor of App Server Y, we were able to increase the TPS on that app server by approximately 100%. Try as they may, though, they were not able to get close to the performance number of App Server X, which were still approximately 300% greater than App Server Y. During the process, we did disclose to the vendor of App Server Y, the performance numbers being produced by App Server X, so that they had something to shoot for.

Again, please note that the deficiencies of bridging from a web server to the J2EE container using a "connector" are amplified by running both the web server and the J2EE app server on the same machine, as we did.

By documenting this experience we do not wish to say or remotely imply that a "connector" architecture is inappropriate in general.

In our discussions regarding this problem with the vendor of App Server Y, they informed us that most of their customers circumvent this problem by:

(a) running the web server on another machine, and/or

(b) running multiple web servers (on other machines) which fan in to the J2EE app server

One could argue that the workaround discussed above (running multiple web server(s) on other machines) is more of a common case than running both the web server and a J2EE app server on the same machine. In fact, the apparent assumption by most J2EE app server vendors that any high performance site would run the web server(s) on separate machine(s) is probably a reasonable one.

That said, it does not mean that the configuration tested in this study is invalid. Far from it. In fact, we have seen it used at innumerable clients.

Furthermore, we must point out that in the absence of a real test of these codebases on configurations suggested above (multiple web servers on multiple machines which fan in to a much smaller number of J2EE servers), the better expected performance of the connector architecture in these configurations is partially speculation on our part. Removing the bottleneck created by the connector architecture will spotlight performance of other areas of the application servers (the EJB container) where we believe from anecdotal experience that App Server Y is very good, and thus may have an opportunity to outperform App Server X, or… App Server X may still be faster (but probably by a narrower margin). We don't know.

Finally, using the multi-machine-fan-in configuration may open up a small can of worms, with people arguing over whether an identical deployment configuration should be used for all application servers, or whether the multi-machine deployment configuration "recommended" by each application server vendor be used. If the vendor recommended best practice is used then the additional hardware might very well double the throughput, if one more server was added, for the other vendors.

For these reasons, it is probably appropriate to interpret the measurements of J2EE App Server X (instead of those of App Server Y) as more representative of the J2EE platform specifications' capabilities. However, for the reasons described above, currently, most app servers displayed performance similar to App Server Y. We ask readers to use their judgment.

Please also see Section 25 for more detailed technical theories on why App Server X outperforms App Server Y in this configuration.

14 Test Lab & Test Software

We used Mercury LoadRunner 7.5 to conduct the performance tests.

The tests were run in a large-scale test lab that consisted of 100 physical client machines capable of generating very high concurrent user loads while ensuring that the clients were not a bottleneck in the system.

A CISCO gigabit backbone was used, with each server configured with two gigabit network cards each talking to a subset of 50 clients.

This setup was designed to ensure that the network was never the bottleneck during testing.

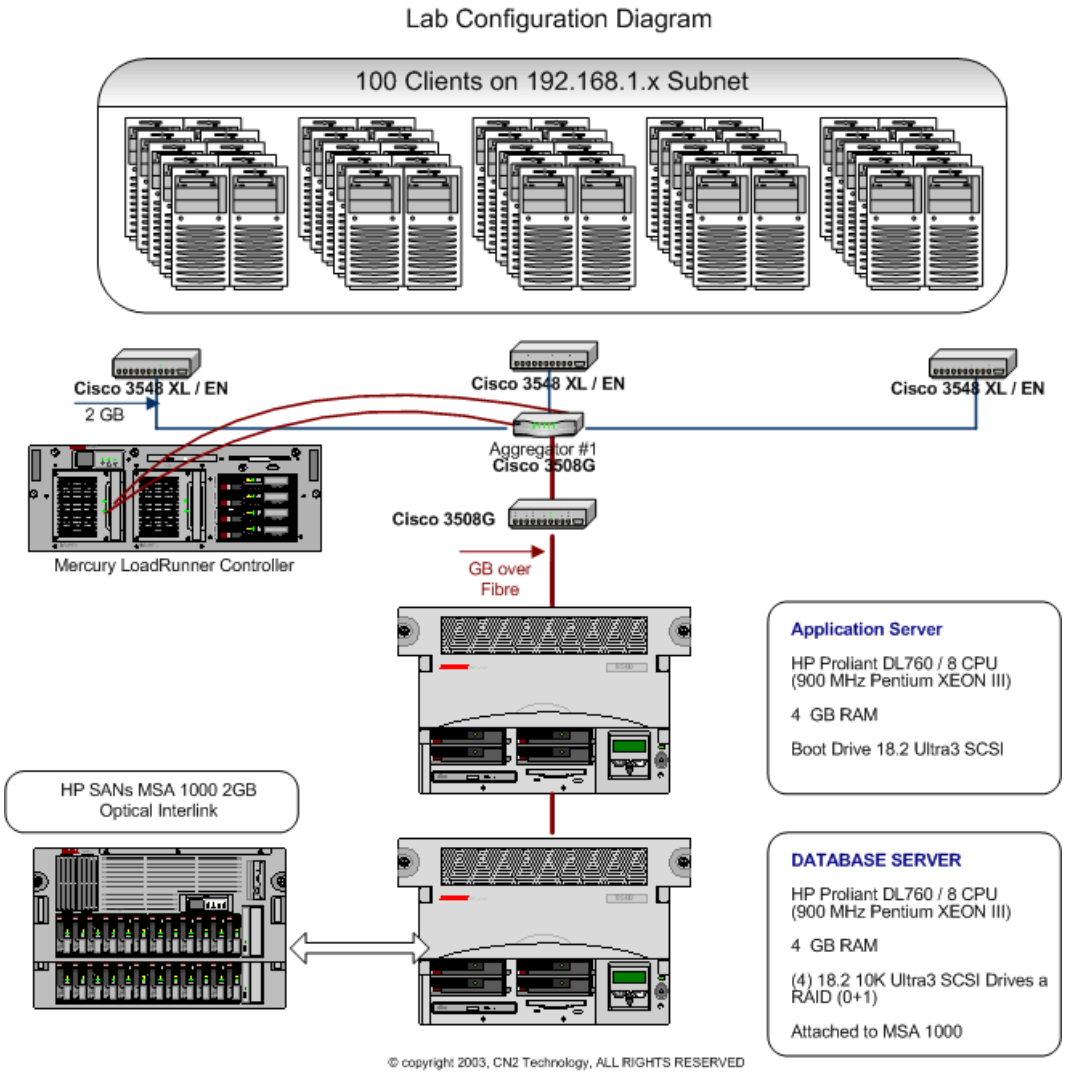
Two separate database servers were also configured on the gigabit network.

The application servers tested were run on Hewlett Packard ProLiant DL 760 Servers, configured with 8 900 MHz CPUs, and 4 GB RAM. The database was also run on Hewlett Packard ProLiant DL 760 server, each with 8 900 MHz CPUs, 3 GB RAM, and fiber optic SANS network controllers attached to fast Compaq RAID SANs storage arrays.

Database, client, and network utilization were monitored during testing to ensure these never became a bottleneck. For all codebases, the tests accurately report the capability of the application server software itself for the applications tested.

The operating system for all tests was Windows Server 2003.

The diagram below shows the lab configuration for all tests:



15 Experiences Tuning the J2EE Middle Tier

15.1 Overview

Experiences tuning the J2EE middle-tier can be broken down into three major categories:

- Tuning the Java Virtual Machine
- Tuning the app server's runtime settings
- Tuning the application's deplo yment characteristics

The rest of this section considers each of these in turn, and then discusses other interesting experiences, including:

- Experiences with Container Managed Persistence v2 (CMP2)

15.2 Tuning the Java Virtual Machine (JVM)

When tuning a J2EE application, a fact of life is that you must also "tune" the JVM. While doing this, a number of factors came into play, including:

- the overall memory requirement of the application server and deployed application
- the speed (pause time) of garbage collection
- the sizing of the various generational areas spaces

15.2.1 Heap Sizes Etc.

When tuning the Java virtual machine, the first parameter that is tuned is the overall heap space allocated, and the sizes for the various generational areas. Extensive testing showed that the default parameters supplied by the Java 1.4.2 server VM (using the –server option) provided almost the best throughput with the exception of a few parameters.

The overall heap granted (default is 64MB) was increased to be 3GB divided by the number of application server instances being run. So, if four application server instances were being run, each would be allocated 750MB; if only two application server instances were being run, each would be allocated 1500MB, and so on.

Recall that these tests were run on a system with 4GB of physical memory and 8 CPUs. Using 3GB of the available physical memory avoided disk swapping, and the available processors could adequately parallelize its allocation.

To increase the predictability of garbage collection the settings -Xms and –Xmx were set to the same value.

Initially, when working with JDK 1.4.1, we needed to increase the value of the permanent generation heap for JPetStore (the 3rd party application used for the J2EE-SERVLETS-JSP category) because the default of 4MB was slightly too small to hold the JPetStore code and required libraries (struts, yoshida, dom, etc.) However, after upgrading to the 1.4.2 JVM this was no longer required due to the increased permanent generation heap size default of 16MB. Tinkering with this size to fine tune it to the minimum possible did not yield any performance gain; that was only to be expected since the 12MB difference in the size between the different JVM releases was so small when compared to the overall amount of heap space allocated to each server instance.

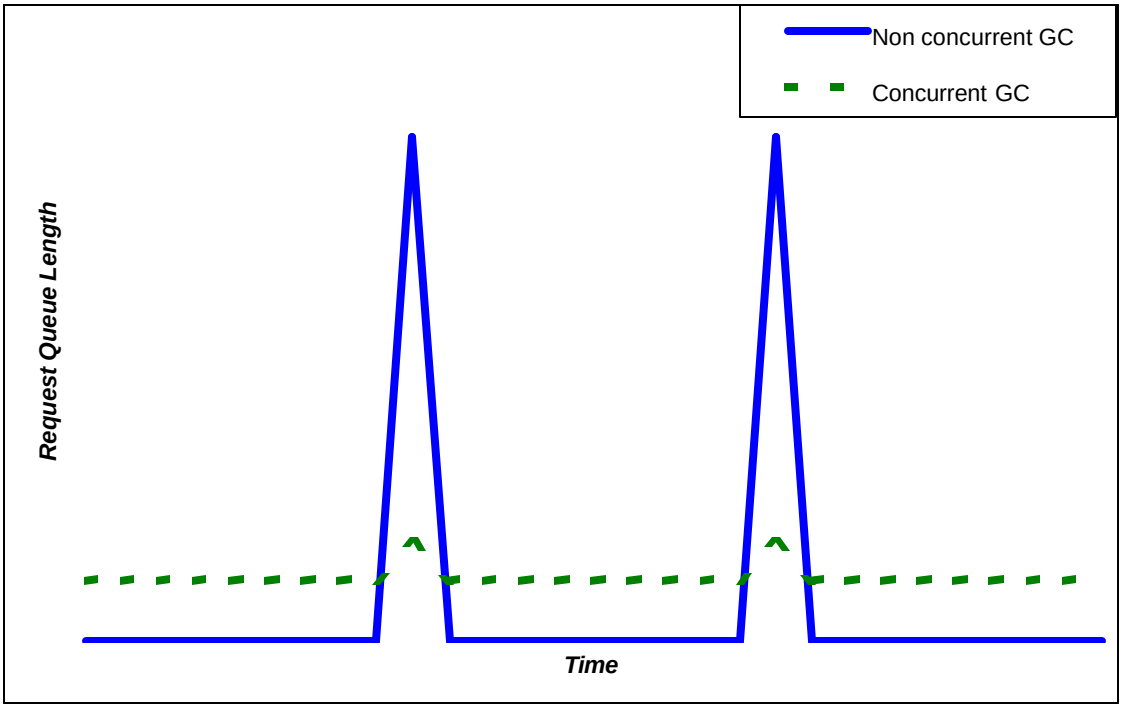
15.2.2 Method of Garbage Collection

Another important parameter that was tuned for the JVM was the method of garbage collection (GC). The question:

> Should we use a concurrent low pause garbage collector (the -XX:+ UseConcMarkSweepGC option) or not?

When using a concurrent GC scheme, extra CPU is consumed by the garbage collector all of the time, and therefore leads to longer queue lengths and response times during the steady state, but leads to much shorter collection times during a full GC. Therefore, the effect is that there is consistently slightly lower performance but only a short pause for a full GC. When using non-concurrent garbage collection, there is higher steady state performance but a significant and painful pause during a full GC.

The pain is severe, especially under high load. During a full GC pause, a server instance is effectively almost off-line; this means that all messages and connection requests have to be queued during that time as shown here in the following figure:



An illustration of the effect on queue length when using concurrent and non-concurrent garbage collection

When running a long test with heavy load (large number of clients and transactions), two factors have to be considered.

First, recall that we have allocated a large amount of heap space… so garbage collection can take a long time, in the order of one, two or even three seconds.

Second, hundreds or thousands of requests arrive per second and all of these have to be queued while the application server performs GC. Often, these can be too many to be queued by the application server and operating system TCP stack.

So the result of using non-concurrent GC, combined with the heap space sizes described above, is that a large number, e.g. tens of thousands, of connection-refused errors during garbage collection periods. Connection refused errors occur when all of the available TCP/IP connection resources are used. The TCP/IP connection resources get used up because the JVM is not processing any requests during a full GC, and all the queues, both application server and operating system TCP stack, become full of requests waiting to be processed.

Thus at a minimum you should, of course, make sure that all operating system and application server parameters are tuned to allow the maximum amount of queuing: these parameters are discussed later in the web server tuning section.

Once these preliminary steps are taken, we must decide whether to:

- reduce the amount of memory so that GC will happen more quickly, *or*
- use concurrent GC

In this case, we chose to use concurrent GC for a number of reasons:

- The application specification allows for the global caching of search results, product, item, and category data. Reducing the amount of memory to a level that allowed for a fast enough full non-concurrent GC would mean either (a) creating slower access data structures to hold the cached data in a condensed form or, (b) reducing the cache size to not hold all the data.
- In the case of mPetStore (the codebase for the J2EE-EJB-CMP2 category), an alternative to changing the application cache structures would have been to reduce the number of EJBs allowed in memory. However, this had a severe impact on performance when then EJB container started to passivate bean instances.

As compelling as these two reasons were, they were only two contributing factors; the major reason and the determining factor for our decision to use concurrent GC was that although non-concurrent GC outperformed concurrent GC during short bursts of testing, over the course of several hours concurrent GC was faster than non-concurrent GC!

The reason for this is that when the application server performs a non-concurrent full GC, the request queues grow and response times go up. The application server then is fully loaded for a period after the GC, processing all the queued requests, trying to catch up; in the mean time, new requests are arriving. As the test goes on more memory is used and the time between full GC's decreases. When the system is at its busiest and the test has been running for a while, the full GC can occur before the queue from the last full GC has been processed leading to a continually increasing request queue length. We found that this saturation point can be significantly delayed by using concurrent GC.

In addition to testing the concurrent mark sweep collector for the old generation, we also examined using the parallel young generation collector (-XX:+UseParNewGC), possible as the application servers were running on an 8 CPU system. Unfortunately we found no advantage one way or the other between this option and the default copying collector.

15.2.3 Other JVM Options

In previous tests, we had found that when using the –server JVM option that reducing the compile threshold (-XX:CompileThreshold), i.e. the number of times a method was executed before it is compiled into the permanent space, could improve performance. In this performance comparison though we found no advantage in this one way or the other. We believe that this is due to the fact that these tests where longer in duration and client invocations more frequent, so that by the time the application server was under stress the compile threshold had already been passed by all methods in the invocation paths.

We experimented with numerous other JVM options. Most other effects were minor and not noteworthy enough for this report.

15.2.4 CPU Affinity

When the system is under high load and the CPU is fully used, CPU contention will occur.

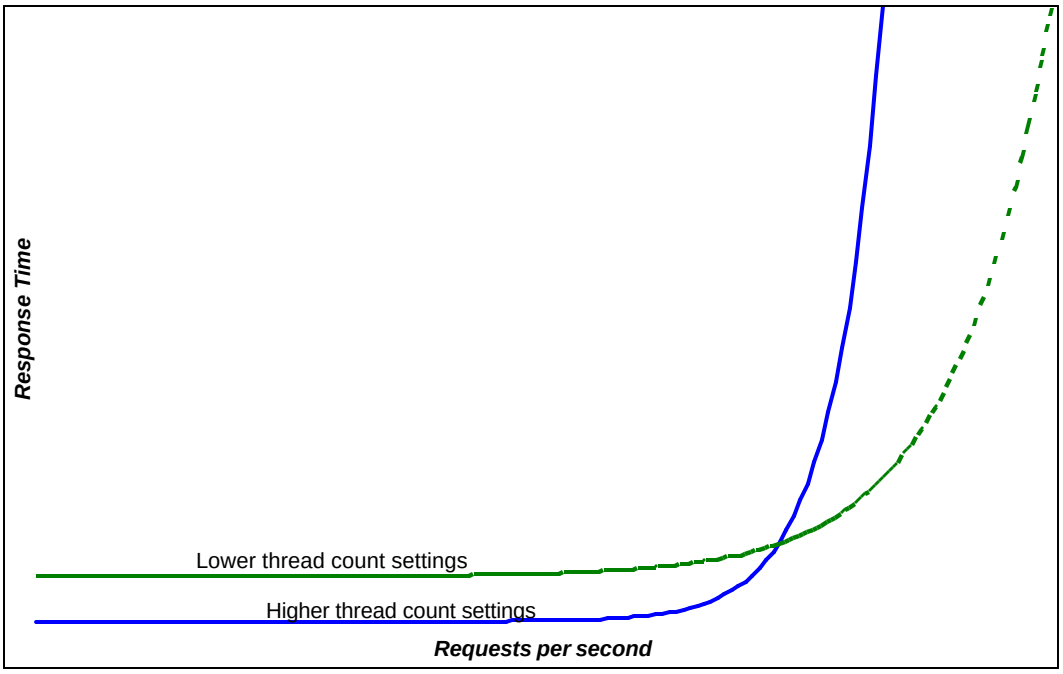
To alleviate this problem, each running application server JVM was assigned to a subset of the available CPUs using CPU affinity. In general this yielded an additional throughput of approximately 100 hits per second compared to running without CPU affinity. For App Server X we used four application server instances and used CPU affinity to assign 2 of the 8 processors to each instance. For App Server Y we used 2 application server instances and assigned 4 CPUs to each instance.

15.3 Tuning the Application Server

15.3.1 Execution Threads

Tuning the number execution threads in the application server one of the most, if not the most, important step in this area.

If the number of execution threads is too low then the CPU will never be entirely consumed, leaving precious resources unused. If the number of execution threads is set too high, then the request queue length and user response times will be lower at low load, but the CPU can become pegged at 100% too quickly, leading to excessive contention, in turn leading to a maximum number of concurrent users that is below the highest possible. This is illustrated in figure below.



An illustration of the effect of execute thread settings on application server throughput

For both of the J2EE applications, in all of the tested application servers, the optimum execute thread setting (i.e. the thread setting that gave the highest number of concurrent users before the CPU was 100% used), was below the vendor default! It was between 9 and 13 threads depending on the application/application server combination. The JPetStore application is more CPU intensive than mPetStore, primarily because of the Java reflection used by the struts framework, and thus used lower thread settings.

15.3.2 Logging

Logging can negatively impact throughput and response time. While it is a good idea to have some logging turned on, so that runtime errors can be fixed, we reduced all application server logging to a bare minimum and disabled any logging that was not needed.

15.3.3 Unused Services

Many, if not all, of the application servers tested came with additional features that were not necessary for this application. Among the services we disabled were:

- Default JMS factory creation
- Automatic session bean passivation timeout. (This was turned off or set to a really large period such as 1 day)
- Cluster broadcasting, where possible
- Example applications
- SSL certificates and unused security

15.3.4 JDBC Tuning

By far the slowest single step in the application is database access. Thus every attempt was made to make sure that the JDBC datasources and underlying connection pools were tuned as far as possible. Although each application server vendor offers a slightly different feature set, many things are common and need to be addressed:

Connection pool size. As with most pools, JDBC drivers offer the ability to grow or shrink the database connection pool size. However, adding, and to a lesser extent removing, connections from the connection pool is an expensive operation. So in most cases it is a good idea to size connection pools similar to the number of execute threads. For example, if there are potentially 8 concurrently executing threads then each connection pool was given 8 database connections with no shrinking or expansion.

Prepared statement reuse or caching. Both the mPetStore and JPetStore applications use JDBC prepared statements extensively and any ability to reuse those prepared statements gives a large performance boost. If the application server provided the ability to size the prepared statement cache we also did that. Although sizing the cache too large does not degrade performance too much, it is best to conserve memory where possible. We found a cache size of 30 worked well, approximately the same as the number of different prepared statements we have in the application.

Query access methods. When running against SQL Server there are two query access methods "direct" and "cursor". Setting selectMethod to direct allows SQL statements to be executed without incurring server-side overhead for managing a database cursor over the SQL statement. We found that the direct mode is the most efficient for executing these queries from these applications.

15.3.5 JDBC Driver Selection

Driver selection is also important. Most JDBC drivers work equally well but may do some things better than others. We spent a few days testing many different drivers (about seven) before settling upon three drivers to use.

For SQL Server we used one driver for all connection types; but for Oracle, we used one driver for XA datasources and a different driver for non-XA datasources.

This last point demonstrates how one JDBC driver vendor may be good at one thing and a different vendor good at another. It also draws attention to the fact that it is worth spending the time to make sure you try many different JDBC drivers to get the best performance and functionality combination.

To reduce CPU contention, thin, all Java, JDBC drivers were used for all databases, applications and application servers.

15.4 Deployment Parameters of the Application Itself

When an application, especially a CMP EJB application, is deployed to a J2EE application server there are a number of parameters that can be configured to optimize performance.

15.4.1 Stateful Session EJBs cache sizes

Stateful session EJBs cache sizes can be configured to allow for the number of beans in the application server cache. For the sake of performance it is best to configure this parameter such that session beans are never passivated to the file system or a database as this adds a great deal of overhead and can reduce performance of the entire application by up to 30%. When deploying the mPetStore application, we repeatedly ran the application at a very high user load (12,000 concurrent users) for a long duration, 14 hours, and used a binary chop technique to determine the lowest number of beans in the cache that would cause no disk passivation. For mPetStore this number was 45,000.

15.4.2 Stateful Session EJBs idle times

Stateful session EJBs idle times allow you to configure the amount of time a bean is idle before it is removed, usually in the case where a client has finished with the session but has not called remove on the bean.

Recall that our web application test tries to model the real world by putting in the following twists:

1) Clients only sign out of the application 50% of the time and this is the point at which their http session and session beans are removed.
2) Clients can be idle for up to 10 minutes before their session is expired.

The mPetStore application maintains a reference to a stateful session bean, ShoppingClientController, in the users http session. ShoppingClientController in turn keeps a reference to another stateful session bean, ShoppingCart. In the case where a client does sign out, the remove method is called on the ShoppingClientController which in turn calls remove on the ShoppingCart, and then the http session is invalidated. i.e.:

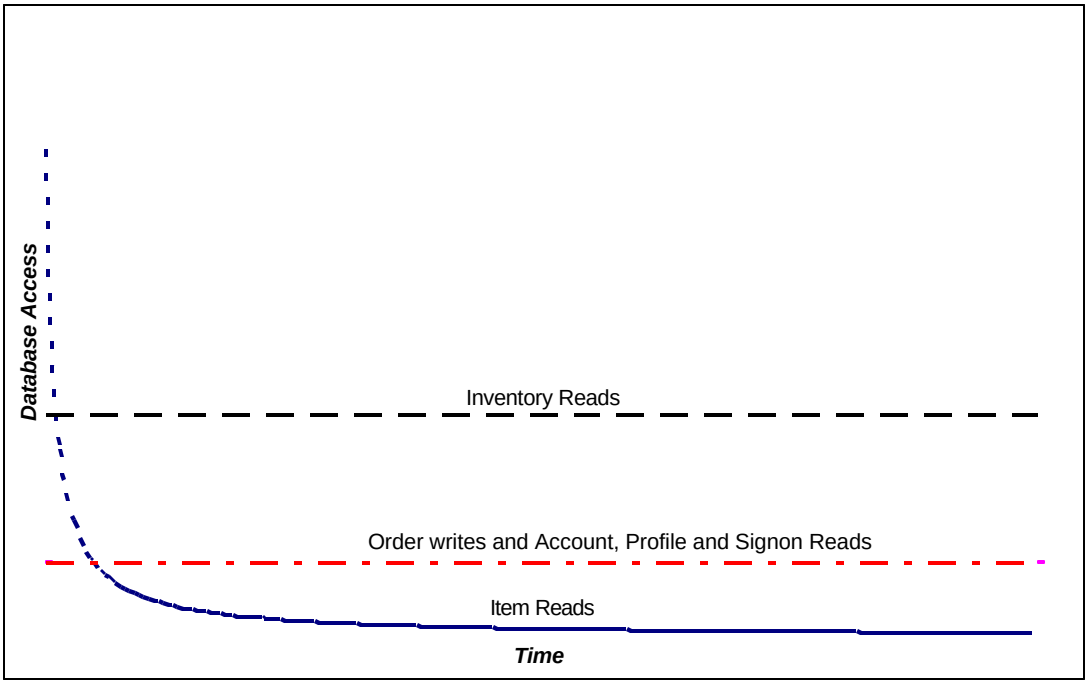
When configuring the idle timeouts, the ShoppingCart is configured for an 11 minute timeout whereas the ShoppingClientController is configured for a 10 minute timeout so that when the ShoppingClientController remove method is called the ShoppingCart will still be present and it will then have its remove method called.

15.4.3 Entity Bean Cache and Pool Sizes

In the mPetStore application there are four classifications of Entity bean:

- **Entities that exist on a per user basis and cannot be cached**, i.e. they are re-read from the database on each access. These are Account, Signon, and Profile.
- **Entities that exist based on the product catalog and can be cached for up to 24 hours**. These are Category, Product and Item.
- **An Entity that exists based on the product catalog and the spec does not allow to be cached**. This is the Inventory.
- The Order **entity that is no use to cache and is usually only created**, i.e. inserted into the database.

Each of these Entities requires different tuning considerations depending upon how it is used and accessed during the duration of the application run. What follows is a chart showing the relative read frequency of each classification.



Relative database I/O frequency over testing time of the different types of Entity bean. Note: Product and Category are not shown, because there are so few rows in each of these tables that they are usually completely read and cached with the first few seconds of testing.

For the Entities that exist on a per user basis and cannot be cached such as Account, etc. we found it optimal to allow one Entity of each type in the application server EJB cache for each potentially connected user that would sign in. Note, although we might run 12,000+ users in a test in the non-transaction heavy scenario only half of those users would sign in, the other half would only browse. We set these cache sizes at 7,500.

For the Entities that exist based on the product catalog and can be cached for up to 24 hours, we set the application server EJB cache to the lesser of: the number of data rows in the table, or the number of concurrent users that may be accessing that data. For Product and Category data these were set to 1,000 and 5 respectively. Once this data is read, the data itself is cached in an application level cache and the entity is never used again for the next 24 hours. The Item entity cache was set at 15,000 allowing each potential concurrent entity transaction to have its own entity instance. There is no need to size this EJB cache to 50,000 (the number of rows in the Item table), because after being read by a user the data is cached in an application level cache.

For the Inventory Entity that the spec does not allow to be cached and exists based on the product catalog, the application server EJB cache is sized to be equal to the maximum number of concurrent entity transactions, which was the number of concurrent users: 15,000. There is no benefit in sizing this to be the same as the number of rows in the inventory table (50,000) since, the previously cached copy must always be reread to get the latest inventory information.

For the Order Entity that is no use to cache and created when a user checks out, the application server EJB cache is sized to be equal to the maximum number of users that might checkout after purchasing items. This number was 7,500, i.e. 50% of the maximum user load.

15.4.4 Other J2EE Application Tuning

A number of other parameters come into play when deploying a J2EE application that can aid in enhancing its performance.

Automatic loading of entity state. When the EJB finder methods are invoked, vendors will often automatically, or allow you to configure an option to, load the entity state at the same time. This is particularly useful when you know that after calling a finder method you will be using data from that entity, i.e. you are not simply calling find to verify existence. Therefore, all beans in the mPetStore application take advantage of this feature whenever possible.

Session timeout. When configuring the web deployment descriptor, the maximum session timeout is set to 10 minutes to comply with the case study specification.

Recompilations of JSPs. In the web deployment descriptor any options to re-load and/or recompile Java Server Pages (JSP), based on changes or timeouts are turned off to eliminate expensive recompilation while the application is running.

15.5 Experience with Container Managed Persistence v2 (CMP2)

15.5.1 Background

After the results from our October 2002 performance case study were published, a large number of members of the community suggested that the J2EE codebase should have used Container Managed Persistence, in particular version 2 (CMP2), rather than Bean Managed Persistence (BMP).

The speculation at the time was that EJB 2.0 CMP Entity Beans could not only perform better than BMP, but also drop the lines of code (LOC) count[1].

This was one of the factors that the expert group who drafted the spec considered, and is likely one of the reasons that category J2EE-EJB-CMP2 is included in the spec.

Therefore, in the current implementation, most of the BMP beans in the old codebase were converted to CMP2 beans.

15.5.2 Experience

Many of the BMP Entity Beans were simple model objects mapped to a single table in the database. Converting these beans to CMP2 was straightforward. We imported the database schema into Borland JBuilder and used JBuilder's EJB wizards to generate the bean classes and deployment descriptors. The converted beans' home interfaces typically contained simple finder methods (besides findByPrimaryKey()) and create methods. Their component interfaces typically consisted of access methods (getters and setters).

One BMP Bean, the Order EJB, followed the Composite Pattern. This bean represented four tables in the database and used sequences to generate primary keys for three of the tables. We successfully decomposed the Order EJB into three CMP2 Entity Beans with Container Managed Relationships (CMR); ultimately, however, we decided to keep the Order EJB as BMP for reasons that follow.

We decided against the CMP 2 version for Order EJB because the SQL statements in the BMP version were more efficient than the SQL statements created by the container. With BMP, we did in one statement what the container attempted to do in three statements. We tested the speed of the final BMP version

[1] Note again that the version of the spec we used considers the LOC count obsolete; more information is available in The Middleware Company Application Server Platform Baseline Specification, http://www.middleware-company.com/casestudy

against (a) the CMP2 version, as well as (b) an alternative BMP implementation that used multiple SQL statements. The BMP Order EJB containing a single SQL statement ran fastest.

Without getting into the details of the Order EJB and the relationships among its database tables (see the specification), the following describes some of the hurdles to converting the Order EJB to CMP 2. The first problem we encountered in converting the Order EJB into multiple CMP Entity Beans, was the lack of support in the EJB specification for automatic primary key generation using database sequences. Many application servers do support this feature and as mentioned above, we did successfully implement the Order EJB as three CMP 2 Entity Beans: Order, LineItem, and OrderStatus. However, since the EJB specification does not support this, we feared that there would be portability issues with the code base.

The LineItem and OrderStatus tables contained compound primary keys composed of the orderid from the Orders table and the linenum column from their own tables. Since the orderid was represented through a CMR field (a link to the Order EJB), the in-memory primary keys for the LineItem and OrderStatus EJBs were mapped to the linenum CMP field. This field represented a unique value since the same sequence generated the values for LineItem and OrderStatus. According to the EJB specification, a compound primary key must contain values from the CMP field list from the deployment descriptor (EJB Specification 2.0, Section 10.8.2). Attempting to use the order CMR field caused EJB compilation errors.

The Product EJB also remained implemented as a BMP Entity Bean. Initially, the Product EJB contained several complex finder methods for performing searches against the database. In one method, a Category EJB was passed as a parameter and the finder method used its primary key to search for products. The EJB Specification is ambiguous about using EJBObjects and EJBLocalObjects as input parameters (EJB Specification 2.0, Section 11.2.7.4). The specification notes that it is up to the container to map EJB parameters to abstract schema types but doesn't seem to enforce this. There was not a container managed relationship between the Category EJB and the Product EJB so a comparison such as the follwing did not work:

```
SELECT OBJECT(p) FROM ProductEJB WHERE ?1 = p.category
```

Note: ProductEJB is the abstract schema type for the Product EJB and `?1` represents the first input parameter (Category EJB type) to the finder method. The following is invalid in EJB-QL:

```
SELECT OBJECT(p) FROM ProductEJB WHERE ?1.id =  p.categoryid
```

Note: `p.categoryid` represents a CMP field on the Product EJB. Later, we changed the method signature to accept the Category EJB's primary key as a String value, allowing us to use the key value in a WHERE clause comparison in EJB-QL.

With that problem solved, a second finder method became the hurdle to CMP 2 conversion. This finder method accepted a keyword for the product search. Although the interface was simple and mapped to EJB-QL, a problem existed in converting the SQL statements of the BMP finder method to EJB-QL. The method used the "lower" SQL operation which, along with the "upper" operation, is not supported by EJB-QL or by vendors. The Product EJB remained implemented as BMP.

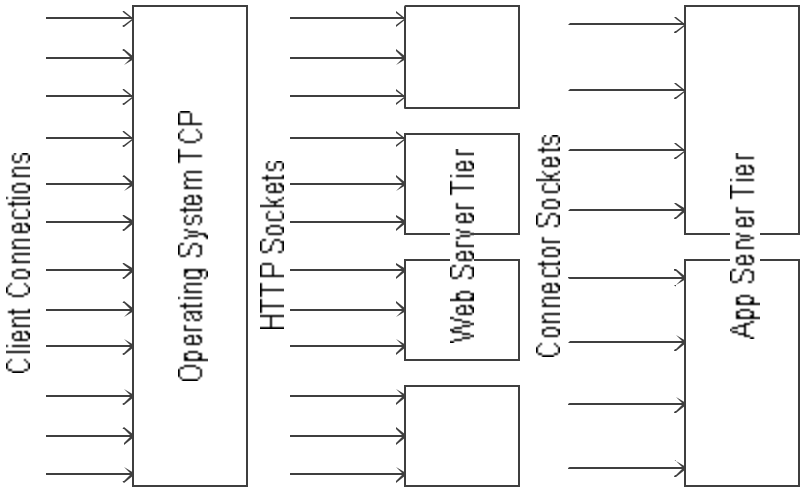
16 Experiences Tuning the Web Tier & TCP/IP Layer

16.1 Overview

Here there are several areas to be considered:

- The operating system & TCP/IP layer

- The Web Server(s)
- The Connector(s) from the Web Server to Application Server



Depending on the application server being used, the web server and application server may be combined into a single process usually removing the need for tuning connectors between the web server and application server.

In order to make sure that all of the application server execute threads are kept busy we generally employed the scheme:

Number of http reader threads ^{3} *Number of connector reader threads* ^{3} *Number of application server execute threads.*

However, this requires careful tuning when the server becomes busy. The scheme relies heavily on the operating system, connectors, and application server to queue requests. For example, if there are too many http reader threads then eventually the queue of requests for the connector will get too large and clients will be unable to connect. Therefore, a the majority of the tuning time in the web server tier is spent making sure that queues have enough depth, and that just the right balance of thread counts is achieved.

16.2 Tuning the Operating System & TCP/IP Parameters

Recall that for all of the application servers we used Windows 2003 as the operating system.

Since the application server layer may pause for reasons described in Section 15, for up to three seconds at a time. Even though three seconds seems short, the volume of http requests was high (up to 1,800 per second), so the main considerations in the operating system layer were:

- That it be able to queue incoming connection requests if they could not be immediately accepted.
- That we provide enough space, and rapid access to it, so that the operating system is able to process TCP data as quickly as possible.

To achieve these goals we changed, or added, the following TCP/IP registry settings. Note the values given here are typical examples only; during actual testing, these were changed for each application/web server combination to ensure maximum throughput with minimum CPU usage.

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

- **MaxUserPort** - Increased to the maximum 65534 to enlarge the number of user ports available.
- **TcpTimedWaitDelay** - Decreased to 60 second to reduce the number of ports in TIMED WAIT state.
- **NumTcbTablePartitions** - Changed to 16, so with an 8 CPU system we could get more concurrent Tcb table access.
- **TcpMaxSendFree** - Changed to 0xFFFF, from 5000, to bump the size of the TCP Header table.

HKEY_LOCAL_MACHINE\SYSTEM \CurrentControlSet\Services\AFD\Parameters

- **EnableDynamicBacklog** - Turned on to allow Windows to dynamically shrink or grow the number of backlogged connections.
- **MinimumDynamicBacklog** - Set to 20, when the number of free connections on a listening end point drops below this value additional free connections will be added.
- **MaximumDynamicBacklog** - Set to 20000, the maximum number of free connections for a listening end point. Note that in a commercial server this may be reduced to prevent denial of service attacks that were not a concern on our closed lab environment.
- **DynamicBacklogGrowthDelta** - Set to 100, i.e. the number of free connections to create when additional connections are necessary. Note that a large value for this setting could lead to a too large free connection allocation.

16.3 Tuning the Web Server Tier

For the J2EE codebases, whenever it was necessary to use an external web server, we used Apache version 2.0.45. Several tests were performed using other web servers, including comprehensive ones with IIS, on the Windows 2003 and 2000 platforms. However, we found that of the available vendor connectors to the application servers, the Apache connectors gave the best throughput.

A great deal has been written elsewhere about tuning Apache performance so we simply list the values that we tuned for this application:

KeepAlive: This parameter was turned off. Although better performance can be achieved by leaving this parameter on, it can lead to starvation because with KeepAlive turned on, requests are passed to the connector using a LIFO mechanism.

ThreadsPerChild: Tuning the Apache threads per child is very similar to the tuning of the application server execute threads discussed earlier. The optimum value is the lowest value at which the CPU can still reach 100% usage. The important consequence of this setting is that this also will effect the number of application server threads; no matter how high the application server execute threads is set, they will never be used if there are not enough Apache threads reading data to keep them busy. We found the best performance was reached by setting the threads per child to 12. Note: There were four Apache instances used so the total number of threads was 48.

ListenBacklog: Apache passes the listen backlog parameter to the socket listen call therefore, this value is MaximumDynamicBacklog ^{3} ListenBacklog ^{3} MinimumDynamicBacklog. In order to allow for more than sufficient queuing in the operating system we set this to 5,000.

Listen: When configuring Apache we found we could get the best throughput with four Apache instances each listening to a separate IP address.

HostnameLookups: This was disabled to avoid unnecessary DNS calls.

LogLevel: Set to "critical". While performing all initial setup and tuning this was set to a much higher level of verbosity, but lowered for actual testing.

SSLEngine: Set to off because SSL was not needed for this comparison.

Although it was tempting to disable access logging, this feature was left enabled despite the fact that it would generate > 5GB of access data in a 24 hour period. In most production environments, access logging would be enabled, and we felt that disabling it would be unrealistic. All platforms, applications, application servers left this enabled.

16.4 Tuning the connectors from Apache to the Application Servers

There are very few parameters that can be tuned in most of the connectors from Apache to application servers, other than the number of connections to the application server, which we made equal to the number of Apache threads.

And we made sure that debug logging was turned off.

17 Experiences Tuning The .NET Middle Tier

.NET tuning was performed primarily by two Microsoft employees, under the observation of The Middleware Company. From our observations of the engineers during tuning, experiences, .NET was stable and reliable. Like all software it was possible to make mistakes but because of the tight platform integration the task of writing, deploying, tuning, running and monitoring appeared to be easy and straightforward.

.NET 1.1 was set up on Windows Server 2003 by enabling the Windows Server 2003 Application Server Role.

The default configuration was further modified as follows for all measurements performed:

- IIS 6.0 was setup with 4 Web sites, each listening on a separate IP address, each setup with the PetShop virtual root. Each Web site was configured with its own Application Pool. Hence, for each measurement performed, 4 separate ASP.NET worker processes ran simultaneously, fully process isolated from the HTTP IIS 6.0 Service.
- Machine.Config was modified to set authentication from "Windows" to "Forms" to match the authentication model of the PetShop application. Likewise, Windows authentication was turned off in IIS for the PetShop virtual roots.

With this common base configuration, there were three application-specific settings in .NET that were adjusted to obtain the best results. The optimal values of these settings depended on the test being performed. These settings were:

- Number of database connections in the ADO.NET connection pool. This is set in Web.Config.
- Number of COM+ objects in the COM+ object pool (COM+ was used for transaction support for order placement business component only). This is set in COM+ component Explorer.
- Number of ASP.NET worker threads. This is set in the ASP.NET Machine.Config file.

The basic tuning approach was to match number of database connections in the pool with the number of worker threads and the number of COM+ objects in the object pool.

One Windows 2003 registry entry was changed:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Http\Parameters

- **MaxConnections** - Increased to the maximum 60000 to enlarge the number of connections available.

The following settings were used for the various measurements:

.NET Web Application Test and Web Services Test with Oracle:

- 4 database connections per connection pool
- 4 COM+ objects per worker process in the object pool
- 4 ASP.NET worker/io threads

.NET 24-Hour Reliability Test with Oracle:

- 8 database connections per connection pool
- 8 COM+ objects per worker process
- 8 ASP.NET worker/io threads

.NET Web Application Test and Web Services Test with SQL Server:

- 8 database connections per connection pool
- 8 COM+ objects per worker process in the object pool
- 8 ASP.NET worker/io threads

.NET 24-Hour Reliability Test with SQL Server:

- 12 database connections per connection pool
- 12 COM+ objects per worker process
- 12 ASP.NET worker/io threads

Several performance monitor (perfmon) counters, including processor utilization, thread contention, ASP.NET request queue size, and garbage collection counters, were used to tune the application and arrive at these settings.

Approximately 3.5 days were spent on monitoring performance of the .NET application while adjusting these settings before arriving at the final settings.

18 Experiences with JPetStore (Codebase for J2EE-Servlets-JSP)

18.1 Review of Key Differences between JPetStore and mPetStore

JPetStore is an open source implementation of the The Middleware Company Application Server Platform Baseline Specification created by iBatis.com. It utilizes Struts and an advanced reflection-based database mapping layer designed to run in a Servlet container without the use of EJBs.

> Note that we worked with Clinton Begin, the author of JPetStore, to eventually remove the reflection-based database mapping layer and use straight JDBC, for two reasons: (1) to make the codebase conformant to the J2EE-SERVLET-JSP category which requests straight JDBC, not an arbitrary database mapping layer, and (2) to improve performance.

mPetStore, on the other hand, utilizes simple JSPs and statically generated database mappings utilizing the Servlet container along with both Session and Entity beans.

What this boils down to is that the original JPetStore made heavy use of reflection but neither invoked any EJBs nor used any static code generation. In contrast, mPetStore uses little to no reflection but heavily depends on EJBs and server generated code.

18.2 The Java Reflection Based Database Access Layer

Using Precise Indepth for J2EE, a production profiling tool based on runtime code instrumentation, what we discovered is that given the high throughput and low invocation times of the system (unloaded transaction response time can be as low as 20ms, including database access time), the cost of reflection is very high relative to the cost of other operations performed during a request. The biggest reflection cost was incurred in the database mapping layer where a large amount of runtime reflection is used to enable an XML file to define the mapping of JavaBean properties to columns in a ResultSet.

As a result of this observation, we worked with lead author of JPetStore to write a replacement database layer based on the DAO objects in the mPetStore, which does not use any reflection. This change produced a marked (approximately 10x) increase in throughput!

However, performance of the modified JPetStore was still significantly lower than for the mPetStore.

18.3 Struts

A second round of profiling revealed high cost in the reflection used to access JavaBean properties in the Struts-based JSP pages. The cost was particularly high on pages that displayed lists in tables, since those pages would have to use reflection on potentially large collections of JavaBeans. Due to time constraints, we were unable to try converting the pages from utilizing Struts property accessing tags to static JSP expressions.

18.4 The Famous EJB Overhead?

What about the famous "EJB invocation overhead" then? From our observations, the cost overhead of invoking and Enterprise JavaBean is actually quite low because invoking the container generated EJBObject and related helper objects are all static method invocations that are handled quite efficiently (and often inlined) by the JVM. Similarly, even the much maligned Entity Bean performed quite well because they were used in cases where they were appropriate (retrieval and display of single rows with significant opportunities for container caching) and bypassed in cases where they would be a hindrance (insertion of a complex set of rows like orders with line items while simultaneously updating inventory).

Because Entity Beans do their type-mapping via static code generation, their invocation overhead is correspondingly low, but their use relieves the programmer from a significant burden in the situations where they are useful.

Does this mean that reflection is to be universally reviled and EJBs used everywhere? No. As with all performance issues, these results are highly specific to the properties of the application being tested. Remember that this application performs very little computation and that the natural, unloaded, invocation time is extremely low. That means that we are quite sensitive to operations that increase the invocation latency. Lower invocation time means that individual requests can be processed very quickly and we need fewer worker threads to keep all our pipelines full. Any significant increase in invocation latency means that we may need more threads to fully utilize our hardware resources and thus increase the concurrency overhead, particularly on synchronization monitors such as those guarding access to writeable caches or database connection pools. In a different application that spent more time performing computations or waiting for the database, we might be less sensitive to the overhead of runtime reflection.

Similarly, we can take away from this the observation that we should not simply discard the idea of EJBs because of blind fear of intangible overhead. Depending on various deployment descriptor and application server tuning settings, EJBs could create significant overhead. What is important is that we understand how EJBs work and use the right bean for the right purpose.

18.5 Caching

Section 19, which describes our experiences with the codebase for the J2EE-EJB-CMP2 category, contains the description of the caching layer we built, which we also used for this category.

19 Experiences with mPetStore (Codebase for J2EE-EJB-CMP2)

This section describes our experiences creating and modifying the codebase for the J2EE-EJB-CMP2 category.

19.1 Changes suggested by the community after the prior case study

We started by incorporating the vast majority of changes suggested by the community, after our last case study.

While these changes were being made, we documented the changes, and the impact of those changes, on The Middleware Company case study website. (Most of the detailed information is still available there).

19.2 EJB Home and Remote interfaces were changed to LocalHome and Local interfaces

One of the most exhaustive changes made in this release of the mPetStore application was to change all applicable EJB Home and Remote interfaces to LocalHome and Local interfaces. This was done mainly as a maintenance exercise to make sure that the code was "up to date".

This was one of the changes suggested by many in the community. Note though, that the Remote interfaces being used previously were being optimized by the application servers so that calls made within the same process became local calls, rather than a remote call, even though the code may be using a remote capable interface! Therefore, as expected, testing has not shown major improvements in performance due to this change.

19.3 Caching: Use EJBs to cache?

In The Middleware Company Application Server Platform Baseline Specification there are a number of types of information that can be cached for up to 24 hours. We did a lot of experimentation to find out the best way to cache this data.

One possible way to cache data, is to use read only entity EJBs with the attitude that an entity is effectively a cache for data in the database. However, this is an imperfect solution for a couple of reasons:

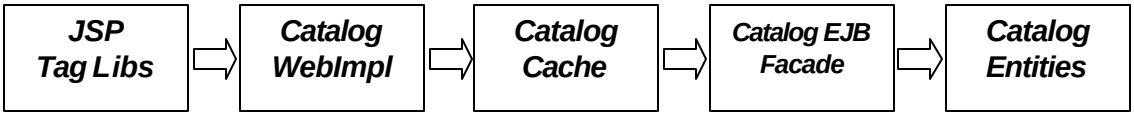
- You cannot search across data in EJBs. So the method findByKeyword, which searches using a case insensitive match with leading and trailing wildcard "%" characters and a SQL LIKE clause, would still have to make a database call. This would leave one of the most expensive database calls in place.
- Whenever you make a call to an entity EJB, the container will still require a transaction to be started and stopped. So, although you are able to get around some database calls you still have the container transaction overhead.

For these reasons we implemented a global application cache.

19.4 Location of Cache

We placed the global application cache in front of the catalog session bean facade.

The mPetStore application has a layer of classes, typically ending in "WebImpl" that are the interface between the web tier and the session bean facade. The logical place to insert the cache has between the WebImpl layer and the session bean facade. Therefore, if we ever spilt the application between a web server and application server the cache could be located in the web server avoiding any remote calls to the application server for cached data. The WebImpl class responsible for the cached catalog data is CatalogWebImpl.



Note that for the JPetStore codebase, EJBs were not allowed. It worked out nicely that we were able to implement almost exactly the same cache between the PetStore data access and SQL layers.

19.5 How Much Data to Cache?

The next decision we made was how much data to cache. There are a number of different cache combinations possible in this application:

- All the categories (in this test there are 5 of these)
- All the products for a category (5 categories with approximately 200 products each)

- All the products (1,000 of these)
- All the items for a product (1,000 products with approximately 50 items each)
- All the items (50,000 of these)
- All search strings (1,000 search words but many, many possible combinations of multiple search words)

One possible strategy was to create separate caches, one for each of the combinations. Another was to create only three or four (e.g., category, product and item, and possibly, search words) and create a set of groupings or references to allow the other combinations to be computed. Obviously, the first approach would require less compute time but more memory, and the second would require less memory but more compute time.

When we looked at these choices we found that the overall memory required for the first approach would be about 16MB; for the second approach it would be 8MB. Because the memory difference was so small, and we aimed to fully utlize the CPU in the middle tier, we decided to use the first approach and save CPU time.

19.6 Cache Algorithm and Data Structures

The specification requires that we use a well known / describable algorithm for maintaining the cache lists—we chose an LRU algorithm. Initially we implemented this algorithm using a LinkedList for ordering and a HashMap for access. But in testing we found that after the Item cache had become populated that the list maintenance was taking an excessive amount of time.

Fortunately, the fix was easy and straightforward. In Java 1.4 a new class, LinkedHashMap, addresses exactly this issue and will automatically handle access order and removing entries if the size exceeds a configured maximum size. Therefore, there was no need to maintain a separate List and Map. To use this class we extended the LinkedHashMap in our class CacheMap and used one of these objects for each of the cache combinations.

19.7 Implementation of SQL Statements relating to Orders

The Order entity EJBs SQL statements are the most important in the PetStore implementations for two reasons:

- The committing of an Order, involving the update of four tables, is the single slowest step in the entire application, and therefore the greatest bottleneck to performance. Any improvement in this step's execution yields enormous benefits to the overall application throughput.

- Although not used in the "normal" running of the application, the reading of the Order and LineItem tables is the single slowest step for the getOrder web service. So, improvement in this step provides a lot of performance improvement to the web service throughput.

Of these two, the trickiest to implement is the Order commit step not only because there are four tables to be updated (Order, OrderStatus, LineItem and Inventory), but also, according to the Specification, the Inventory may exist in a different database from the other three!

19.7.1 Writing Orders

We know that in JDBC a prepared statement will give better performance than a regular statement, and we know that executing a single JDBC batch will give better performance than multiple JDBC executions. However, you cannot batch together multiple heterogeneous prepared statements; you can only batch homogeneous prepared statements or heterogeneous regular JDBC statements.

Therefore, the updates to the Orders database (Orders, OrderStatus, and LineItem) cannot use a JDBC batch with prepared statements; updates to the Inventory database can, as there is one statement of the following form for each line item in the Order:

UPDATE INVENTORY SET QTY=QTY-? WHERE ITEMID=?

So we reviewed a few possibilities:

- Create a single JDBC batch from JDBC statements for Orders database updates and a single JDBC batch of prepared statements for Inventory database updates.

- Perform multiple JDBC updates from prepared statements for Orders database updates and a single JDBC batch of prepared statements for Inventory database updates.

- Use a single JDBC batch of prepared statements for Inventory database updates. Programmatically create a single compound prepared statement that updates all Orders database tables. For example, in Oracle, this would be a PL/SQL block like this:

```
BEGIN
   INSERT INTO ORDERS VALUES(?,?, ... ,?);
   INSERT INTO ORDERSTATUS VALUES (?, ... ,'P');
   INSERT INTO LINEITEM VALUES (?, ?, ?, ?, ?);
   INSERT INTO LINEITEM VALUES (?, ?, ?, ?, ?);
END;
```

  …where there would be a variable number of the INSERT INTO LINEITEM lines, one for each line item in the order.

Although the last option is more work (developer-time-wise and CPU-wise for the string handling), the execution speed of this option was over 30% faster than either of the others. Hence this was the method we used in the final version of the application.

19.7.2 Reading Orders

As we mentioned above, the reading of the Order and LineItem tables is the single slowest step for the getOrder web service. So, improvement in this step gives a lot of performance improvement to the web service throughput.

Since the order has one header row, in the orders table and for each order row multiple line items it is intuitive to do one SQL select to read the order header information, and then another to read the order line information. However, another way to do this is to join the line item and header information into a single SQL statement. Obviously, there are pros and cons to both approaches:

- If we read the Orders table and the LineItems table each separately then we perform two SQL selects

- If we read all the data in one SQL joined select, like this:

  SELECT ORD.userid, ORD.orderdate, ..., ORD.cardtype, LI.linenum, LI.itemid, LI.quantity, LI.unitprice FROM ORDERS ORD, LINEITEM LI WHERE ORD.orderid=? AND LI.ORDERID=ORD.ORDERID

  …then we have a much larger result set, with all the order header information present for every line item.

We found that it was much faster (approximately 25% faster) to perform the single select than it was to perform two selects. In the final implementation of mPetStore and JPetStore we used the single joined select for reading orders.

20 Experiences with msPetShop (Codebase for .NET-C#)

The codebase for the .NET-C# category was provided by Microsoft.

Because the authors of this codebase have written an MSDN article[1] that discusses their experiences and decisions in creating this codebase, this section does not go into the same detail, but rather, summarizes the key changes.

20.1 Key Changes

This section summarizes at a high level the key changes that Microsoft was asked to make (and did make). Most of this feedback can be found in its original form on TheServerSide.com community site.

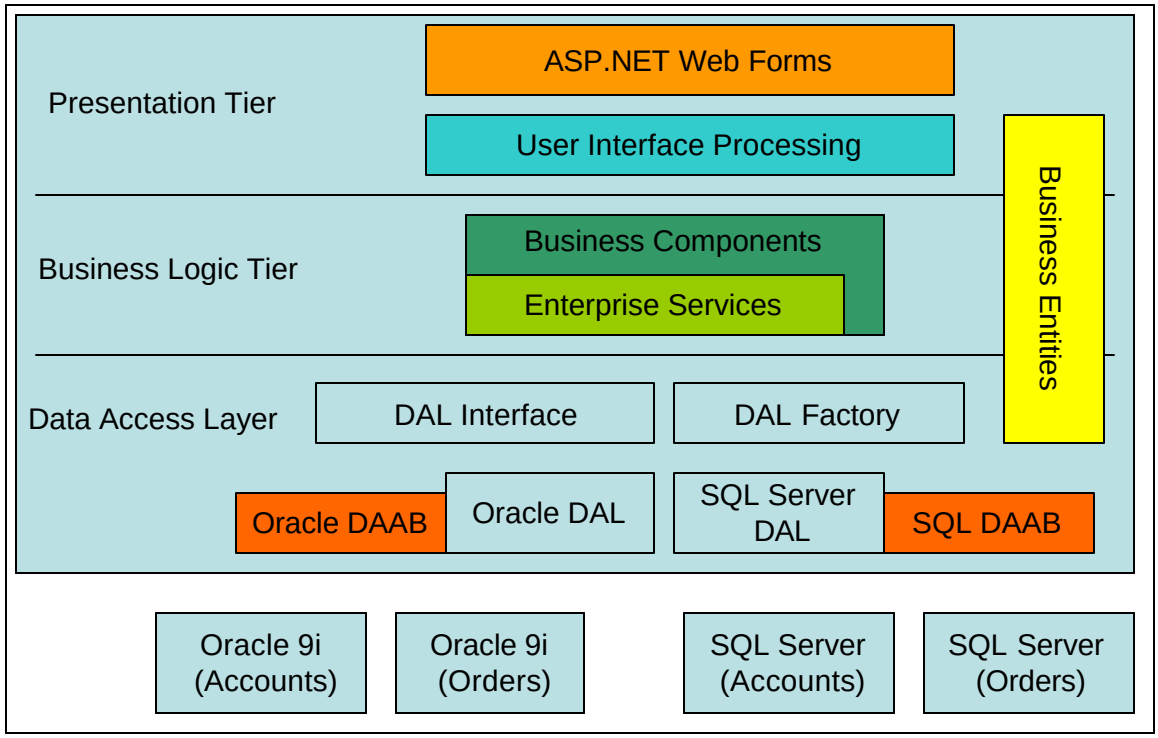
Many of the key changes from the version of msPetShop used in The Middleware Company's October 2002 performance case study (version 2.0), were to answer critics who complained that it was not good architecture.

To that end, the following changes were made.

1) Complete abstraction of middle and data tiers, so the middle tier (Business Logic Layer or BLL) is 100% blind to the backend DB and exactly the same when running with Oracle or SQL.
2) A Data Access Layer (DAL) Factory layer which dynamically loads the correct DAL for Oracle or SQL depending on the configuration settings. Hence, you only deploy one app, and based on simple config options in web.config you tell it which databases are SQL and which are Oracle. It can work in mixed mode as well so the Orders database can be Oracle when the main database is SQL and vice versa, complete with COM+ distributed transaction across the heterogeneous DBs.
3) De-reference Web/UI tier in BLL layer. The business layer no longer references session context; so there is full abstraction of BLL from both UI and Data tiers.
4) Creation of a Model layer that contains object definitions for the common DB elements such as Products, Orders, Items, Inventory, etc.

[1] MSDN article on the .NET-C# codebase: http://msdn.microsoft.com/library/en-us/dnbda/html/PetShop3x.asp

20.2 Architecture Diagram of the new .NET-C# codebase



20.3 Database Performance vs. Database Portability

Among the key changes in the msPetShop codebase is their approach to the tradeoff between database performance and database portability.

The Middleware Company Application Server Platform Baseline Specification required Microsoft to provide an implementation of the application that supported multiple databases without code changes (specifically for the performance case study, Oracle and SQL Server).

When designing the database access mechanism for the application, .NET developers usually face a choice as to which database providers they should use: (a) the generic OLE-DB managed provider or (b) the database-specific, performance-optimized .NET managed providers, such as the SQL Server and Oracle managed providers provided with the .NET Framework 1.1.

Microsoft chose to build the application using the database-native .NET managed providers because they were aware that this submission would be used in a performance case study.

For an analysis of the difference in performance between the managed providers and the generic OLE-DB providers, readers should refer to *Using .NET Framework Data Provider for Oracle to Improve .NET Application Performance*[1], which shows that the vendor-specific providers can perform two to three times better than the equivalent OLE-DB provider.

By choosing to use database-specific access providers, Microsoft made the tradeoff that they would need to write a separate data access layer for each database platform.

[1] http://msdn.microsoft.com/library/en-us/dndotnet/html/manprooracperf.asp

And to simplify the deployment of these database access classes Microsoft used the Factory Design Pattern as outlined by the GoF, with reflection being used to dynamically load the correct data access objects at runtime. The factory is implemented as follows: a C# interface is created with a method declared for each method that must be exposed by the database access classes. For each database that they want to support, they must create a concrete class that implements the database specific code to perform each of the operations in the interface or "contract." To determine at runtime which concrete class to load, they create a third class which is the factory itself, which reads a value from configuration file to determine which assembly to load, using reflection.

The scheme is described in more detail at the MSDN article referenced above.

20.4 Stored Procedures

Stored procedures were a hotly debated issue in previous case studies.

The Expert Group that formalized The Middleware Company Application Server Platform Baseline Specification chose to disallow stored procedures for reasons that are discussed in the feedback doc on the case study website.

Although Microsoft recommends the use of stored procedures as a best practice to their customers, Microsoft agreed with us that for the purposes of such studies, it is best to not muddy the waters with stored procedures. They discuss this issue in the MDSN article referenced above:

> *"Normally we would recommend to customers that they use stored procedures to access tables in the database. However, to make the best use of the financial investment that you have made in your database software and hardware, developers will tend to optimize the SQL used in their applications for their specific database engine regardless of whether the SQL is in a stored procedure or generated in the middle tier. A good example of this is generating unique numbers or identity numbers, as all databases support their own particular mechanisms for doing this, and so the SQL used to generate the unique number tends to be specific to the database being used. There are always alternatives, but they do not perform as fast as the proprietary solutions. With the .NET Pet Shop we took the conscious decision not to use stored procedures in the application as this might be seen to be an unfair advantage for the .NET solution in the Middleware Benchmark[1]. In reality the difference in performance is small because the application is relatively simple and most of the SQL statement execution plans are cached in the database. However, for purposes of the Middleware Benchmark specification, which disallows the use of stored procedures even to wrap simple SQL statements, the .NET Pet Shop 3.0 uses no stored procedures."*

20.5 Caching

.NET has two built-in mechanisms for caching.

One of them is ASP.NET Output Caching, which can be very useful to improve performance by caching dynamically generated HTML pages or fragments, so that they do not need to be re-generated again. The Middleware Company Specification, however, disallows Output Caching since that can simply measure the efficiency of the http caching mechanism.

The other built-in mechanism is .NET Object Caching (a.k.a. Cache API), which allows you to cache data in the middle tier, using the .NET framework internal caching engine. As described in Section 0, this kind of caching is allowed and encouraged by the Specification.

[1] The choice of the word "benchmark" is theirs, not ours. Again, for reasons described in Section 4, we choose to call it a performance case study, not a benchmark.

The .NET codebase leveraged this API to perform caching conformant to the Specification. The MSDN article referenced above provides more detail, with code examples, on how the caching was implemented.

During performance tuning, the Microsoft developers were able to observe the behavior of their cache by using perfmon.

20.6 Itemized List of Changes

20.6.1 Changes made in going from version 3.1 to 3.2

1) Renamed the order components to match their function rather than implementation
2) Created a configuration tool to help with application setup
3) Add option to have two different DALs in use, one for the web and one for the orders system
4) Add encryption to database connection strings
5) Inclusion of a web input string cleaner in the web application
6) Move the process flow and navigation aspects of the UI into a separate set of classes under Web\ProcessFlow
7) Cached the profile query results for the banner control
8) Set all instances of AutoEventWireUp to 'false'
9) Create a constant for session keys

20.6.2 Changes made in going from version 3.0 to 3.1

1) Changed all static methods to instance methods
2) Fixed the version number in assembly.info to a specific number
3) Add pre and post build steps to the solution, this runs gacutil and regsvcs utilities

20.6.3 Changes made in going from version 2.0 to 3.0

1) Created a Data Access Layer (DAL).
2) Created a Common project namespace for Model items
3) Removed references to System.Web in Components project
4) Changed the app from server/custom controls to user Web User Control, pager is still a server/custom control.
5) Changed the solution/project structure and the application namespaces
6) Added factories and interfaces for the DAL layer
7) Added DAL implementation for Oracle
8) Added [Serializable] tag to all model classes
9) Converted all public member variables to private and created properties
10) Changed OrderInfo to have fixed array type of LineItemInfo. Removed need to add XML type mappings in class

21 Experiences Tuning Oracle 9i

21.1 Goal

Although it is somewhat of a simplification, the main goal of database tuning in this context of middle-tier performance study, simply stated, should be:

> To ensure that for all the tests, the database is NOT the bottleneck.

This is very important, because if the database is a partial bottleneck (in other words, there are times when the middle tier "waits" for the database tier), then measurements of the middle tier performance under this condition do not measure the full performance potential of the middle tier, and are thus are only partially valid.

Interestingly, the presence of this condition (of the database being a [partial] bottleneck), can make app servers with very good performance (in the deployment configuration used for the test) look "not as good as they really are" while making app servers with bad performance (in that same deployment configuration) look "not as bad as they really are."[1]

There are a number of litmus tests one can run on middle tier machine(s) and database machine(s) to determine whether or not the middle tier is waiting for the database tier.

The remainder of the section describes the various steps we needed to take, to achieve this simple goal.

> After our tuning efforts, and before we made any performance measurements that we'd report here, we took great care and did verify that the database was not the bottleneck.

Now, we knew that despite all the optimizations, the application's functionality of committing an Order to the database would by definition be database intensive. Therefore one of the sub-goals was to optimize this functionality as much as possible

21.2 Time Spent & Experts Involved & Tools Used

To ensure that we were not missing any database optimizations, we engaged the services of two database tuning experts: an experienced database tuning practitioner, Ron Finch, and a renowned Oracle tuning expert who has worked for Oracle for 16 years and runs a popular Oracle performance tuning website.

The full biographies of both database tuning experts are provided at the end of this section.

At times they used Precise's InDepth for Oracle, combined with InDepth for J2EE to diagnose performance issues.

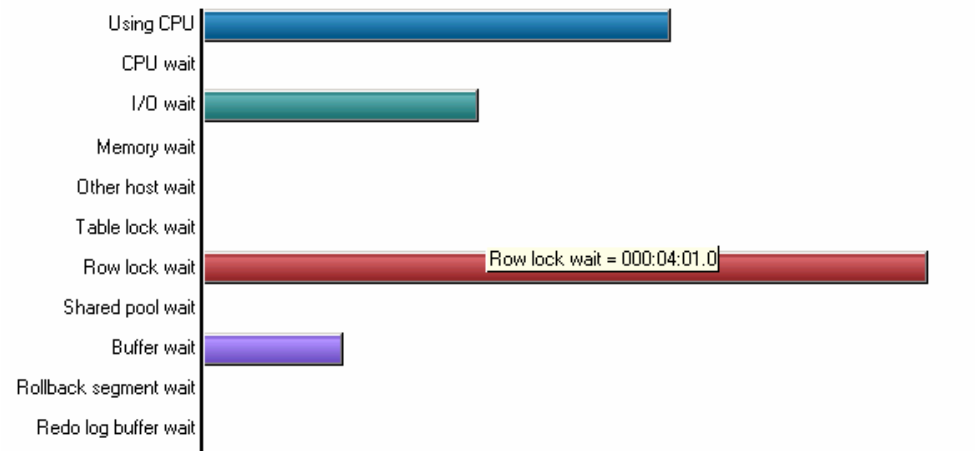
Approximately two man weeks, spread out over the duration of the project, were spent on database tuning.

21.3 Issue: Row Lock Wait Condition

When the experts begain, their initial examination showed that there were a couple of resource issues with the Oracle database.

[1] To take this a step further, it could be argued that for most real world business applications, the database often is the bottleneck! Thus it could be argued that for most enterprise applications, the full performance potential of the middle-tier app server is irrelevant!

You can see in figure below that Row Lock Wait and I/O wait are contributing heavily to the time spent processing the database requests.



A snapshot of relative resource usage in the Oracle database

The first of the areas to be addressed was the rowlock wait condition. After examining the top SQL statements and those contributing to the wait state they found that the main table responsible for this was Inventory.

This is not a surprising result since the Inventory is the most queried table in the application and, with the exception of LineItem, the most updated table. All other tables are either read only, read mostly, or write only. Inventory is the only table that is heavily read and write.

It is not possible to completely remove this wait as rowlocks are an essential part of preserving data integrity. However, it is possible to tune so that the rowlock is held for the least amount of time possible. The first step was to make some alterations to the Inventory table and primary key index:

- PCTFREE is increased to reserve more space for updates, and to prevent rows from migrating when they are updated.
- INITTRANS is increased so that there are more transactions list entries available and Oracle will not have to wait for an entry or wait for an entry to become available.
- BUFFER_POOL KEEP is set on both the table and index to retain the Inventory elements in the database buffer cache and avoid having to wait for physical disk IO.

To implement these changes the following alter table DDL was added:

```
ALTER TABLE INVENTORY PCTFREE 90
STORAGE (BUFFER_POOL KEEP) CACHE

ALTER INDEX PK_INVENTORY
REBUILD PCTFREE 90 INITRANS 16
STORAGE (BUFFER_POOL KEEP)
```

Inventory updates occur inside of a transaction that involves other tables and looks similar to:

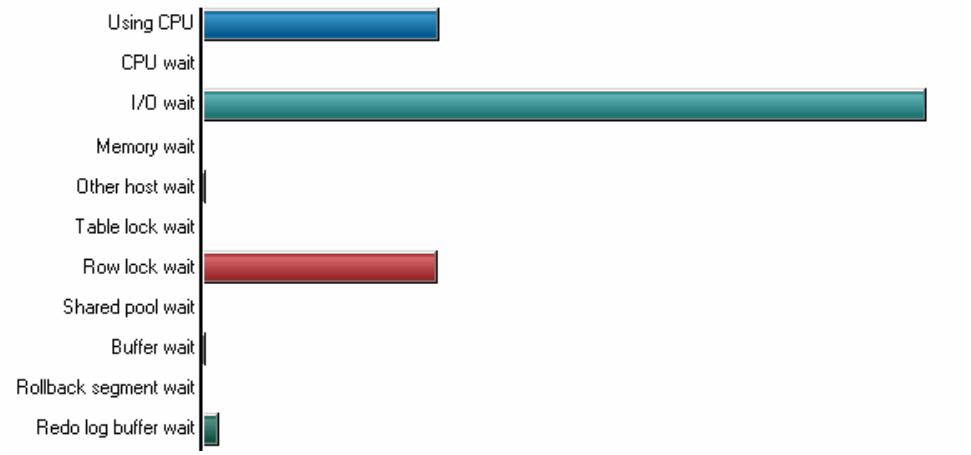
```
BEGIN TX {
        Get next order id sequence number
        Insert into Order
```

```
        Insert into OrderStatus
        foreach line item in the order {
                Insert into LineItem
                Update Inventory
        }
} END TX
```

So, the row lock on Inventory is dependent not only on the Inventory table but also on the speed of the entire transaction. Thus, it is also necessary to make similar changes to the Order, OrderStatus and LineItem tables with the following DDL:

```
ALTER TABLE LINEITEM PCTUSED 10
STORAGE (BUFFER_POOL KEEP) CACHE

ALTER INDEX PK_LINEITEM INITRANS 16
STORAGE (BUFFER_POOL KEEP)

ALTER TABLE ORDERS PCTUSED 10
STORAGE (BUFFER_POOL KEEP) CACHE

ALTER INDEX PK_ORDERS INITRANS 16
STORAGE (BUFFER_POOL KEEP)

ALTER TABLE ORDERSTATUS PCTUSED 10
STORAGE (BUFFER_POOL KEEP) CACHE

ALTER INDEX PK_ORDERSTATUS INITRANS 16
STORAGE (BUFFER_POOL KEEP)
```

After making these changes the database was examined again and we could see that the rowlock wait condition had been significantly reduced:



A snapshot of the relative resource usage in the Oracle database after changes to the order update tables to alleviate the rowlock wait bottleneck.

21.4 Issue: Amount of Physical Disk I/O

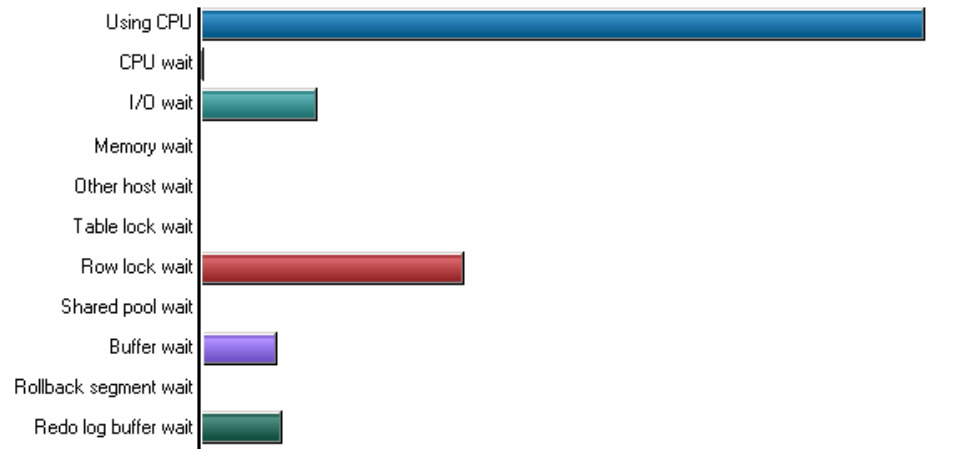
The next condition to be addressed was the amount of physical disk I/O and resulting wait. On close examination we found that this was due to both disk reads and disk writes.

Disk reads were due to an undersized buffer cache. Although this is quite an intensive application when running under heavy load the actual amount of database data is quite small, as far as most databases go. So, reducing the number of disk reads was relatively easily fixed by increasing the Oracle buffer cache size by 128MB.

Disk I/O wait time for writes was due mainly to writes to the redo logs and undo tablespaces. The steps taken to reduce this were:

- The undo tablespace was spread across all available physical disks.
- The redo log group was spread across all available physical disks.
- Although the application is transaction intensive all of the transactions are very short lived. Because of this there is no reason for the undo retention not to be reduced. This was reduced to 30 seconds.

These changes greatly reduced the amount of disk wait time in the database as can be seen here:



A snapshot of the relative resource usage in the Oracle database after changes to reduce the disk I/O wait bottlenecks.

21.5 Biography of Database Performance Consultant Anjo Kolk

Anjo Kolk is Chief Oracle Technologist with Precise Software Solutions. Before joining Precise, he worked at Oracle Corporation for over 16 years mostly in the RDBMS Performance area and Oracle Parallel Server area. During those 16 years he lived and worked in the Netherlands, Ireland, Japan and the United States. The last couple of years he was working for the Oracle Server Performance Group and later worked with different ISVs (Baan. SAP and Amdocs etc.). During that time he has also worked with many different large customers (like NTT Docomo and other Telco's) all over the world to improve performance of their systems and implementations. While working on these performance issues he developed a tuning methodology called YAPP (Yet Another Performance Profiling method). In this method Oracle systems are tuned based on response times. He has taught many Oracle internals classes all over the world. He is also the author of the well known 'Oracle7 wait events and enqueue' white paper and the website www.oraperf.com. Many authors of different Oracle tuning books have made references to this white paper and website.

21.6 Biography of Database Performance Consultant Ron Finch

Ron Finch is an IT professional with over 10 years experience in the industry. Ron has experience in application, database and systems programming, database administration and systems administration. In these roles he has been involved in the development, deployment and support of large-scale environments in education, health care and financial services. As a Systems Engineer and consultant with EMC, he supported some of the world's largest manufacturing, retail, banking and e-commerce customers and managed engagements deploying Oracle and SQL Server applications. Ron is now a Systems Engineer for Precise Software Solutions, the industry-leading provider of enterprise application performance tuning solutions. In this capacity, he is responsible for pre- and post-sales product support, implementation and consulting.

22 Web Application Test Results

22.1 Description of the Test

This benchmark contained a 50/50 mix of two scripts:

- The first script simulated a user simply browsing the site
- The second script simulated a user looking for and purchasing items from the site.

In the browse-only script, the simulated user would:

1) Go to the site home page
2) Perform three (3) free text searches on 1,000 products, clicking the next button on each search result page.
3) Examine the products in a category three (3) times.
4) Examine the details of a particular product item three (3) times (this includes a real-time read of product inventory counts).

In the browse and purchase script, the simulated user would:

1) Perform two (2) free text searches on 1,000 products, clicking the next button on each search result page.
2) Sign on to the site using a random user id from one of the 100,000 available users.
3) Add two (2) random items, from 50,000, to the shopping cart.
4) Checkout, verify the account information and place the order for the contents of the cart.
5) Half of the time the user would sign out at this time and half of the time not such that 50% of sessions were closed, while 50% were abandoned such that the application server would need to handle closing them down, with a session timeout set for 10 minutes of inactivity. This was felt to be a reasonable simulation of the actual activity on a real e-commerce site.
6) Perform one last free text search after purchase.

22.2 Results

The charts below show the results of testing the web application test scripts with increasing user loads for the five codebases against both databases, and the maximum throughput they achieved. Figures 1 and 2

show the results for the applications running using Oracle 9i. Figures 3 and 4 show the results for the applications running using Microsoft SQL Server.

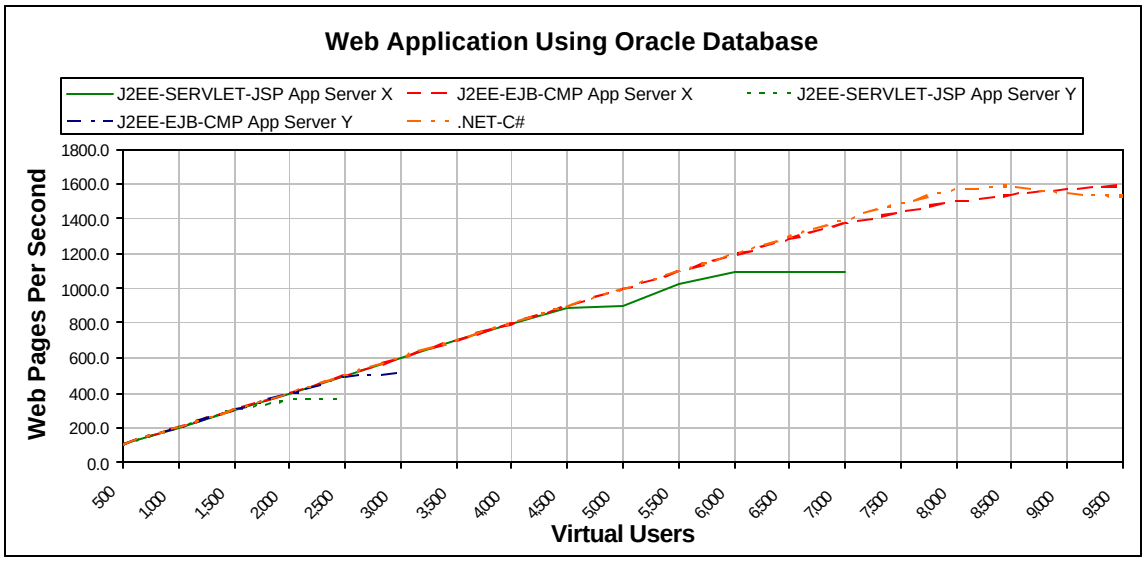


Figure 1, throughput, web pages per second, increase as user load increases running the web application codebases using Oracle 9i database.

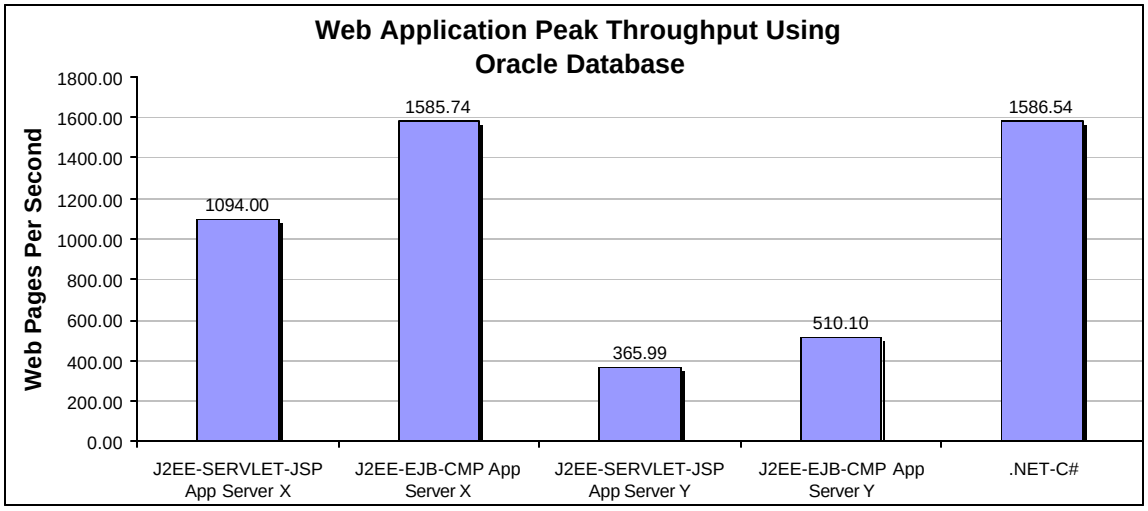


Figure 2, the maximum throughput achieved during the web application tests using Oracle 9i database.

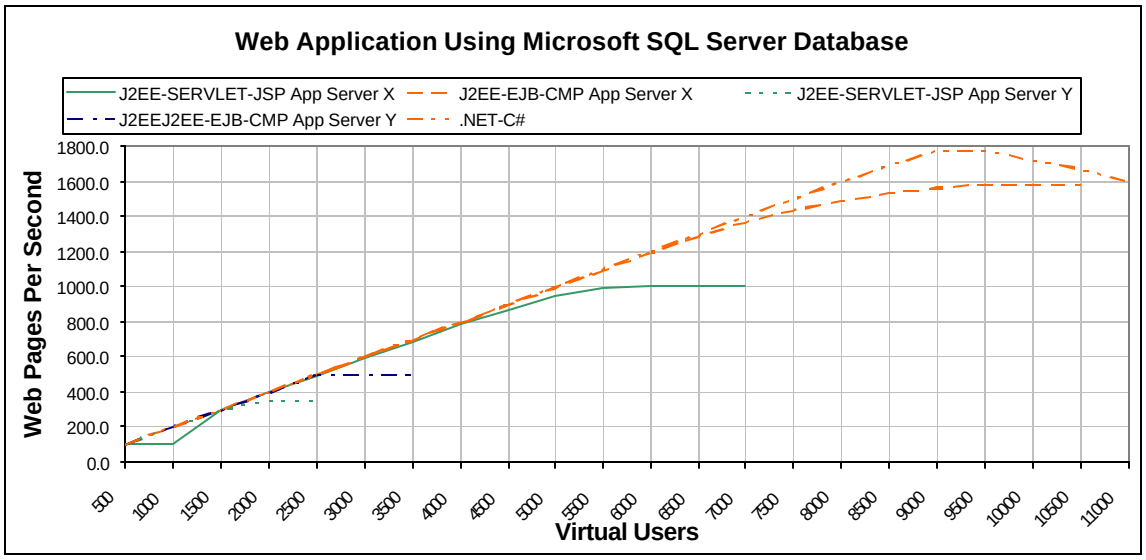


Figure 3, throughput, web pages per second, increase as user load increases running the web application codebases using Microsoft SQL Server 2000 database.

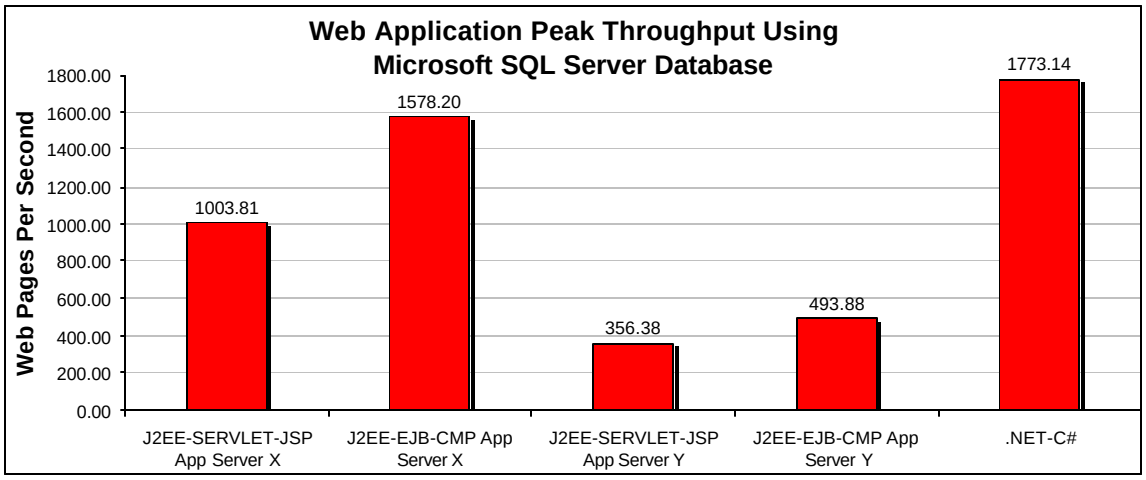


Figure 4, the maximum throughput achieved during the web application tests using Microsoft SQL Server 2000 database.

23 24-Hour Reliability Test Results

23.1 Description of the Test

The 24-Hour Reliability Test was designed to thoroughly test the application server ability to handle high loads of transactions; in this case, the writin g of orders.

The script consisted of simulated users doing the following:

- Signing on to the site using a random user id from one of the 100,000 available users.

- Adding a random item, from among the 50,000 available, to the cart and immediately ordering the item. This is performed 100 times each after login.
- Performing the check out process, 100 times for each login. The check out process includes verifying the account information and placing the order for the contents of the cart
- Signing out

The database used for each codebase was the one it appeared to perform most reliably and conveniently with, in the Web Application Test, and other informal tests. For both the J2EE codebases the 24-Hour Reliability Test was run with Oracle 9i. For the .NET codebase the 24-Hour Reliability Test the chosen database was Microsoft SQL Server 2000[1]

23.2 Sustainable Throughput for the Test

The test was run on an 8-CPU application server at a user load that achieved peak, sustainable throughput for each codebase tested. The values for sustainable throughput for each configuration are shown here in table 1, and figure 5 below.

Codebase & Application Server	***Transactions (Average Web Pages Processed) Per Second***	***Throughput (Average Orders Placed) Per Second***
The J2EE-SERVLET-JSP codebase on J2EE App Server X	1025	204
The J2EE-EJB-CMP codebase on J2EE App Server X	1150	228
The J2EE-SERVLET-JSP codebase on J2EE App Server Y	392	78
The J2EE-EJB-CMP codebase on J2EE App Server Y	451	90
The .NET-C# codebase on Microsoft .NET Using SQL Server	1136	226
The .NET-C# codebase on Microsoft .NET Using Oracle	848	168

Table 1, the 24-hour sustainable throughput averages of: All web pages per second, and specifically the Order Commit web page.

[1] The testing requirements were to run this test against only one database however; Microsoft chose to run .NET-C# twice using both Microsoft SQL Server 2000 and Oracle 9i.

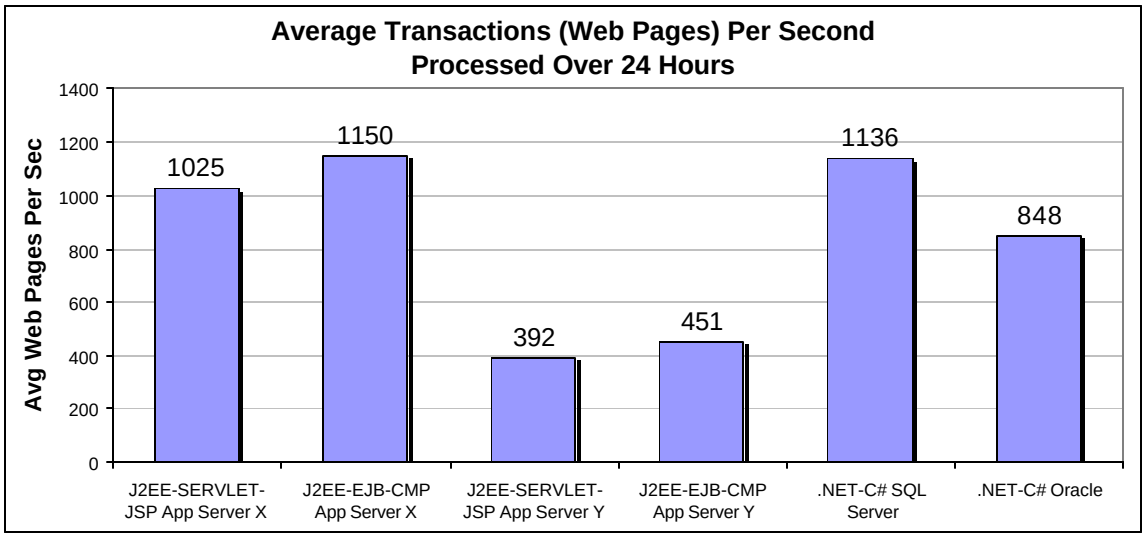


Figure 5, the average number of web pages processed per second over a 24-hour period.

24 Web Services Test Results

24.1 Description of the Test

Since Web Services have been extensively promoted as the new frontier of application servers, The Middleware Company Application Server Platform Baseline Specification includes a description of a Web Service that must be implemented by codebases.

The Web Service itself is specified such that it would provide a realistic test case to illustrate application server performance inclusive of base object activation over SOAP, as well as serialization of simple and complex data types/objects into XML.

The functionality of the Web Service in the Specification is to take as a single input parameter an OrderId, and then to perform a database lookup of that order, returning an Order object as XML to the calling application. The Order object itself contains both simple data types (string, integers, and decimals) as well as an array (representing individual order detail lineitems).

Please refer to the Specification for the WSDL, and other details about the Web Services functionality expected of the codebases.

In this test, clients send a SOAP request for a random order, of 10,000 possible orders. Each order contained five repeating (5) line items in the returned order object. This test puts high stress on a single application server hosting the Web Service, and is designed to illustrate how well the application server performs as a Web Service host, servicing SOAP requests from thousands of concurrent users.

The database used for each codebase was the one it appeared to perform most reliably and conveniently with, in the Web Application Test, and other informal tests. For both of the J2EE codebases the Web

Services Test was run with Oracle 9i. For the .NET codebase the Web Services Test the chosen database was Microsoft SQL Server 2000[1]

24.2 Results

The charts below, Figures 6 and 7, show the results of testing the web service test scripts with increasing user loads for the five codebases against their selected database, and the maximum throughput they achieved.

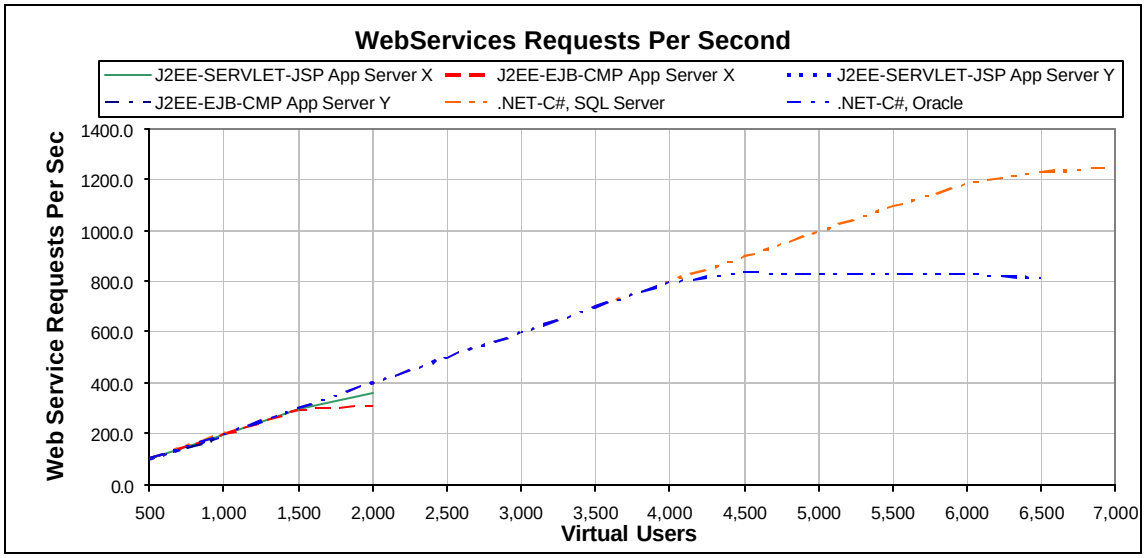


Figure 6, showing the throughput, web service requests per second, increase as user load increase for the various configurations.

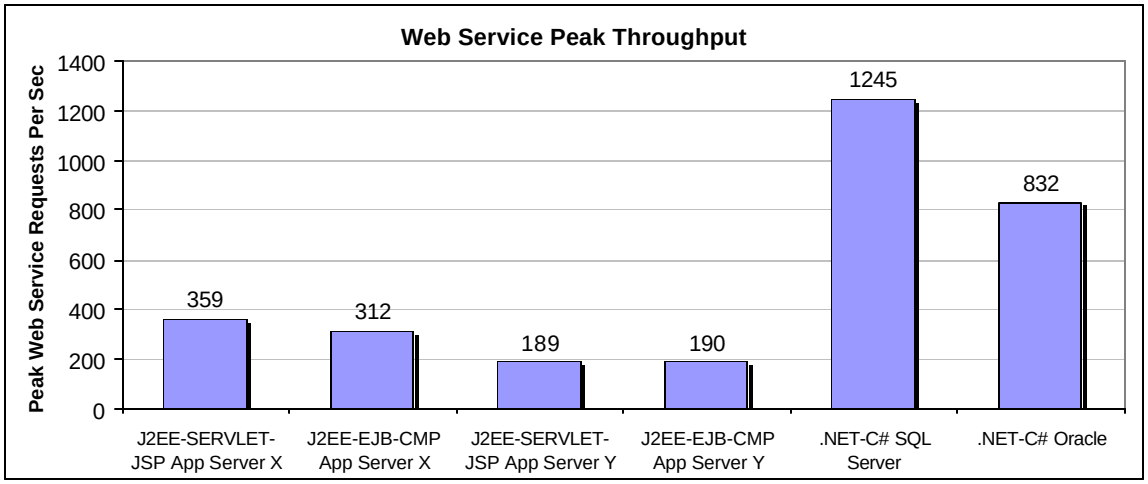


Figure 2, the maximum throughput achieved by each configuration during the web service tests.

[1] The testing requirements were to run this test against only one database however; Microsoft chose to run .NET-C# twice using both Microsoft SQL Server 2000 and Oracle 9i.

25 Technical Theories on why App Server X far outperformed App Server Y (in our configuration)

This section contains some theories and speculations on why App Server X was so much faster than App Server Y ---again we remind you--- in our particular configuration. Refer back to Section 13, for a more high level discussion.

25.1 First Things First: the application is I/O intensive

Like most enterprise applications, the application described in The Middleware Company Application Server Baseline Specification is an I/O intensive application.

The application itself performs very little logical computation. Essentially, it receives a request from the web, maps it to a SQL statement, executes the query against the database, marks up the results in HTML, and sends it back to the client. The two main requirements of this application are that it must generate few to no errors (< 1.0%) and handle as many concurrent users as possible while maintaining a transaction response time less than 1.5 seconds for any operation.

These specification requirements translate to a number of key technical constraints:

- **We cannot drop connections or stop accepting connections until our system is at its maximum load**. We cannot allow for probabilistic "leakage" of connections even if it results in higher overall throughput.
- **Request processing need not be very concurrent**. Users do not interact with each other, nor do they perform long running operations. In fact, there is basically no state shared between users and very little state shared between a set of requests from a single user. This means that we don't need to be handling many requests *simultaneously* as long as we can handle them *quickly*.
- **We need to accept sockets rapidly and be able to hold large numbers of connections open**. As stated above, this application's core function is to push I/O: reading and writing from network streams. And since we want to push the maximum number of simultaneous users (> 10,000) against a single multi-CPU machine, the application server must be able to efficiently handle accepting sockets rapidly, and holding the accepted sockets open for a potentially long time. So we need to accept sockets rapidly (so as not to drop any connection requests) and be able to hold a large number of socket connections (> 10,000) open.

25.2 Theoretical Performance Models

This section is influenced by Matt Welsh's work on SEDA[1] and related topics from his PhD thesis.[2]

Welsh's research suggests that the most scalable approach to architecting I/O handling in an application server uses an asynchronous pipeline of worker stages, each fed by a queue, using some form of non-blocking I/O API to accept, drain, and fill network sockets. All of these are critical to gaining high-performance server scalability.[3]

[1] http://www.eecs.harvard.edu/~mdw/proj/seda/
[2] http://www.eecs.harvard.edu/~mdw/papers/mdw-phdthesis.pdf
[3] See Welsh's thesis, in particular, compare Figure 3 on page 18 to Figure 6 on page 25 and Figure 7 on page 33.

However, in the servlet environment, there are limitations in what an application server can do. A given worker thread must follow a servlet invocation from request to response; the call cannot be split into stages. The streams inside the HttpServletRequest and HttpServletResponse objects must be standard blocking Java streams. Can J2EE achieve this? The answer is apparently, yes, but it is non-trivial.

Looking at the load test results and comparing them to the experimental results in Welsh's paper, a number of parallels can be observed. Notice how at maximum server utilization, although latency increases, App Server X exhibits little to no reduction in overall server throughput. This behavior is consistent with the behavior one would expect from an event-driven non-blocking I/O based server. In contrast, App Server Y shows the behavior of having both throughput and latency increase (along with a large number of dropped connections) at maximum server utilization, a characteristic consistent with a threaded server.

The first question one may have, is if these differences manifest themselves only at peak load, why do we see such a marked difference in performance? Recall that the criteria of the test is to achieve as many virtual users while maintaining a transaction response time of less than 1.5 seconds. Well, the normal, non-peak response time is under one tenth of a second. By the time transaction response time approaches even one second, our server has reached maximum resource utilization and requests must queue. For a server that performs non-blocking accepts and then throws the socket onto a queue, there is no problem. However, if the accept loop blocks or otherwise stops accepting when all worker threads are occupied (relying on the operating system's accept backlog queue) a significant performance hazard is created.

If the number of worker threads is fixed at a low number (say 8), the accept loop will begin blocking before our server is fully utilized because it will become quite sensitive to the timing of the incoming connections. If a large number of connections coincidentally arrive in the same window of a couple of milliseconds, some of them will be rejected even though there could be no requests arriving in the next window. This causes an additional cascading problem because the unserviced connection requests will cause the operating system to begin dropping SYN packets to prevent the application from being overloaded. When the server drops SYN packets, the client's operating system will retransmit the packet. However, operating systems employ an exponential backoff in their SYN retransmit timers, delaying the retries. Since in our performance case study, the number of simulated clients increases linearly with respect to time, the resulting increase in latency is exponential. The operating system does provide a queue when you listen() on a port (in Java this happens when you instantiate the ServerSocket object) so that you don't drop connections while you're processing a given accept(). By default, this queue is set to a fairly low number (around 200 on Windows 2003, FreeBSD, and Linux) but it is possible to increase that value in the kernel. Under Windows, it involves activating the DynamicBacklog feature in the registry and then setting a number of parameters to determine when the feature activates and what the maximum backlog is. The net result of this is that App Server X is not sensitive to this issue because of its accept queuing behavior. App Server Y, on the other hand, is likely to have been severely hampered by its lack of deep queuing here, both when front-ended by Apache and when accepting requests directly to it's internal http server (although the performance degradation when going directly too its internal http server was much more severe).

If the number of worker threads is increased to a level high enough such that App Server Y can handle each simultaneous request in its own thread, the thread count becomes high enough that the number of threads themselves creates significant overhead. In Java, we begin to see this effect once we start to increase the number of threads over 12 or so. The fewer threads, the better. In the ideal case, we'd only need 1 thread per CPU, but with the effects of various I/O and memory pipelines, we usually need a few (perhaps 2x) more to maximize use of multiple CPU's.

Besides threading, the I/O API itself is an equally important scalability factor. The standard blocking-accept and blocking-stream based API in Java is subject to considerable scalability hazard. When the client connect rate is very high, the blocking accept() loop generates significant overhead over a non-blocking approach. However, that alone won't solve the problem. (The Jetty servlet engine contains a JDK 1.4 NIO adapter that accepts connections using non-blocking I/O. However, after accepting the connection, it shifts the connection into blocking mode and reads from it.) The standard Java I/O API routines don't read from

sockets in the manner most efficient for the operating system. This can be seen in a "top" or "perfmeter" like tool when examining how much processor time is spent in "system" rather than "user" cycles. "System" operations are generally system calls to the operating system's API, which typically include all the different ways of reading from sockets. Most operating systems have several different ways of handling sockets and their associated data streams. The relative portion of CPU cycles spent in "system" as opposed to "user" operations can give us an idea as to how much time our application is spending waiting for I/O operations. At first, it is easy to dismiss the issue and blame it on the operating system. However, the application server must share some of the blame; because of how it uses the different API's available from the OS makes a huge difference in performance.

What we can observe, is that the method used by App Server X to read data from the network is much more efficient than App Server Y uses. It seems likely that App Server X is using some kind of 2-stage pipeline. The first stage uses a thread pool whose dedicated task is to accept connections and drain/fill the associated sockets using some efficient non-blocking I/O API. The second stage actually contains the J2EE worker thread pool, which executes the servlet code but doesn't directly read and write from the socket. Between these two stages is some kind of queue that handles the request/response data going between them.

This paragraph in particular contains some serious speculation. We are basing it on detailed observation of one connector, but generalizing the language to cover all similar connectors. Application Servers such as App Server Y provide an Apache module that dispatches HTTP requests to a matching internal connector over a network connection. From what we can tell, the Apache module does not contain its own queue and thread pool. Therefore, when all the internal worker threads are busy, the Apache processor threads that have accepted an excess number of connections will have to wait their turn to have those requests serviced by the module. The net result is that we get a slight amount of queuing behavior when we set the number of Apache processor threads higher than the number of internal worker threads. However, the depth of that queue is approximately the same as the number of processor threads, which is limited to a few hundred at most because of concurrency overhead. This limit is far too low given our user load. The manner in which I/O is handled in the application server poses another issue. While Apache and the application server retain persistent connections to each other, thus eliminating the need for Java to handle a high accept() rate, the connector is still using standard Java I/O. While there is some debate over this issue, preliminary investigation seems to indicate that standard blocking reads/writes using the streams within the java.net.Socket object incur higher overhead than using a properly crafted native handler that uses non-blocking I/O system calls.

The net result is that App Server Y has much more overhead per request than App Server X stemming from its less efficient utilization of system resources. In a system that spent more time performing computation in the application server tier or waiting for computation to occur in the database tier, or in a different deployment configuration, we might see much less difference between the two application servers. However, since this application primarily exercises the ability of the application server to push I/O between tiers, and the entire middle tier runs on one 8-CPU machine, the nuances in I/O handling architecture make themselves much more apparent.

After writing this section we met with the engineer who worked on the I/O subsystem for App Server X, who confirmed many of our speculations about App Server X.

26 Appendix: Data: Web Application Test Result Data

The results, showing average client response times, and throughput, for the web application test. Table 2 gives the data for the Oracle 9i database, and Table 3 the results for the Microsoft SQL Server 2000 databases.

	J2EE Servlet-JSP App Server X		***J2EE EJB-CMP App Server X***		***J2EE Servlet-JSP App Server Y***		***J2EE EJB-CMP App Server Y***		***.NET-C#***	
Virtual Users	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second
500	0.003	99.7	0.001	99.8	0.005	99.7	0.006	99.7	0.003	99.8
1,000	0.003	199.6	0.001	199.8	0.008	199.3	0.006	199.4	0.000	199.6
1,500	0.005	299.0	0.001	299.5	0.028	298.0	0.009	298.5	0.000	299.7
2,000	0.006	398.9	0.002	399.3	0.670	355.6	0.095	393.9	0.001	399.4
2,500	0.007	498.6	0.004	498.4	1.801	366.0	0.046	494.4	0.001	499.0
3,000	0.013	597.3	0.004	598.1			0.851	510.1	0.001	599.1
3,500	0.023	695.2	0.006	698.0			2.020	472.6	0.001	699.0
4,000	0.021	795.4	0.007	797.1					0.002	798.6
4,500	0.045	890.4	0.009	897.0					0.002	898.3
5,000	0.017	899.0	0.012	995.9					0.003	998.2
5,500	0.174	1021.8	0.024	1092.5					0.004	1097.4
6,000	0.493	1094.0	0.024	1192.2					0.005	1196.8
6,500	1.015	1091.3	0.053	1284.2					0.008	1295.9
7,000	1.451	1089.0	0.090	1372.2					0.013	1393.7
7,500			0.198	1441.1					0.028	1489.0
8,000			0.330	1498.0					0.088	1568.3
8,500			0.541	1535.2					0.338	1586.5
9,000			0.734	1577.0					0.806	1542.5
9,500			1.293	1585.7					1.174	1527.9
10,000			1.414	1587.8					1.476	1531.9
10,500			2.492	1578.0					1.779	1535.2

Table 2, response times and transactions (web pages per second) for the web application using the Oracle 9i.

	J2EE Servlet-JSP App Server X		*J2EE EJB-CMP App Server X*		*J2EE Servlet-JSP App Server Y*		*J2EE EJB-CMP App Server Y*		*.NET-C#*	
Virtual Users	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second
500	0.022	99.8	0.006	99.8	0.007	99.6	0.006	99.6	0.001	99.9
1,000	0.026	99.5	0.008	199.5	0.011	199.2	0.006	199.3	0.001	199.8
1,500	0.023	298.1	0.007	299.1	0.025	297.9	0.008	298.7	0.001	299.4
2,000	0.030	396.7	0.007	398.5	0.630	353.6	0.014	397.9	0.001	399.5
2,500	0.034	495.5	0.008	498.2	1.970	356.4	0.084	491.3	0.001	499.0
3,000	0.079	591.5	0.009	597.6			1.041	493.9	0.001	599.0
3,500	0.062	690.4	0.012	697.6			2.059	491.8	0.001	699.0
4,000	0.106	782.1	0.012	796.4					0.001	798.6
4,500	0.151	872.1	0.015	895.8					0.001	898.4
5,000	0.266	946.8	0.041	990.1					0.002	998.4
5,500	0.537	990.6	0.026	1092.5					0.002	1097.4
6,000	0.965	1000.9	0.036	1189.1					0.003	1197.3
6,500	1.427	1003.6	0.055	1283.4					0.004	1296.7
7,000	1.904	1003.8	0.126	1362.9					0.005	1396.1
7,500			0.223	1432.5					0.008	1495.2
8,000			0.369	1486.0					0.013	1593.0
8,500			0.546	1530.8					0.024	1688.5
9,000			0.759	1560.8					0.065	1773.1
9,500			0.992	1578.2					0.355	1769.8
10,000			1.338	1570.4					0.796	1719.4
10,500			1.637	1571.9					1.263	1669.4
11,000									1.854	1593.8

Table 3, response times and transactions (web pages per second) for the web application using Microsoft SQL Server 2000.

27 Appendix: Data: 24-Hour Reliability Test Result Data

The results, showing throughput, for the 24-hour reliability test. The results show the throughput for the various configurations at regular intervals during the 24-hour period.

Test Time	J2EE Servlet-JSP App Server X Web Pages/Sec	J2EE EJB-CMP App Server X Web Pages/Sec	J2EE Servlet-JSP App Server Y Web Pages/Sec	J2EE EJB-CMP App Server Y Web Pages/Sec	.NET-C# Web Pages/Sec
00:00	963.06	1035.23	401.66	429.65	1115.02
00:34	1065.10	1163.96	407.66	451.07	1188.39
01:08	1063.05	1138.71	405.26	450.45	1168.13
01:42	1063.71	1156.31	390.74	450.79	1146.20
02:16	1064.29	1160.22	404.37	455.02	1129.03
02:50	1062.84	1157.03	399.14	454.60	1133.73
03:24	1063.65	1154.23	392.96	454.84	1129.21
03:58	1060.60	1148.61	397.40	452.26	1126.31
04:33	1063.60	1147.89	400.91	454.85	1134.72
05:07	1064.98	1152.77	396.57	448.82	1134.97
05:41	1063.62	1148.70	399.81	448.40	1131.71
06:15	1062.98	1147.07	395.33	450.10	1133.42
06:49	1063.74	1148.78	399.41	451.18	1131.21
07:23	1063.77	1153.12	395.57	453.29	1135.52
07:57	1061.16	1150.48	400.81	449.08	1132.00
08:32	1064.19	1149.53	386.10	453.16	1130.34
09:06	1063.62	1148.03	387.88	452.45	1134.64
09:40	1015.20	1155.17	401.50	449.59	1128.09
10:14	988.44	1150.25	396.13	451.54	1129.01
10:48	990.48	1146.95	387.67	448.46	1135.70
11:22	989.77	1147.63	395.77	452.20	1127.80
11:56	989.52	1149.32	385.88	449.10	1131.09
12:30	989.11	1148.88	393.74	453.99	1128.04
13:05	989.46	1150.22	375.73	451.44	1145.58
13:39	990.01	1146.21	394.74	449.26	1141.32
14:13	990.67	1148.79	383.46	452.45	1134.46
14:47	992.69	1156.49	390.59	451.13	1144.32
15:21	989.74	1154.53	391.64	450.76	1130.05
15:55	991.67	1157.56	392.68	451.79	1143.96
16:29	990.13	1154.01	391.01	450.50	1142.08
17:04	990.85	1154.00	396.29	451.58	1126.47
17:38	991.47	1157.10	397.88	454.04	1139.37
18:12	990.86	1157.15	375.22	451.77	1142.35
18:46	990.66	1156.39	389.36	454.18	1139.62
19:20	991.21	1153.99	387.28	453.16	1134.42
19:54	990.27	1156.45	386.44	448.83	1131.30
20:28	989.42	1156.11	387.66	451.74	1134.31
21:02	996.48	1153.94	386.75	454.27	1138.45
21:37	1024.49	1154.93	369.36	451.89	1134.62
22:11	1054.85	1158.94	388.37	452.33	1134.45
22:45	1067.81	1158.85	397.65	454.07	1133.55
23:19	1069.14	1156.48	384.57	448.88	1133.57
23:53	1072.82	1167.16	388.00	455.89	1144.46

Table 4, transaction rates (web pages per second) at regular intervals for the 24-hour reliability test.[1]

[1] Only Microsoft's .NET-C# SQL Server 2000, their chosen database for the 24-hour reliability test, results are shown here.

28 Appendix: Data: Web Services Test Result Data

The results, showing average client response times, and throughput, for the web service test.

	J2EE Servlet-JSP App Server X		***J2EE EJB-CMP App Server X***		***J2EE Servlet-JSP App Server Y***		***J2EE EJB-CMP App Server Y***		***.NET-C#***	
Virtual Users	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second	Average Response Time	Web Pages per Second
500	0.009	99.8	0.014	99.6	0.108	98.2	0.023	99.3	0.001	99.9
1,000	0.015	199.1	0.018	199.0	0.299	189.0	0.318	187.7	0.001	199.8
1,500	0.025	298.1	0.060	296.0	3.004	188.2	2.904	189.6	0.001	299.4
2,000	0.567	359.0	1.405	312.0					0.001	399.4
2,500	2.159	348.9	3.121	307.5					0.002	498.9
3,000									0.004	598.5
3,500									0.003	698.5
4,000									0.005	797.8
4,500									0.008	897.0
5,000									0.011	995.5
5,500									0.020	1092.9
6,000									1.220	1184.4
6,500									0.179	1227.7
7,000									0.479	1245.2
7,500									0.914	1242.4
8,000									1.420	1245.7

Table 5 , response times and transactions (web pages per second) for the web service test[1]

[1] Only Microsoft's.NET-C# SQL Server 2000, their chosen database for the Web Service test, results are shown here.