

**Performance Analysis of J2EE-Based
Software Architectures and
Application Servers**

Felix Eichhorn
(orb@gmx.de)

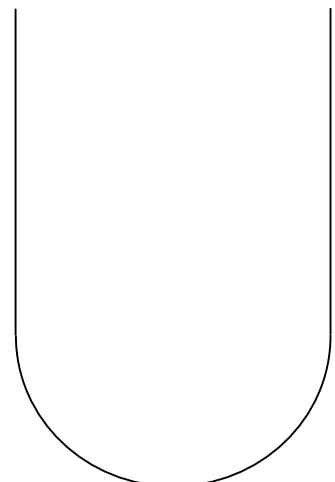
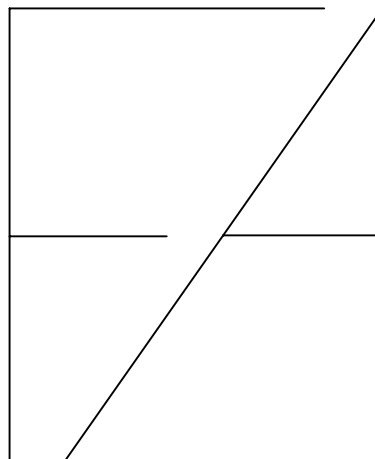
April 2001

SA-I4-2001-06

Studienarbeit

Institut für Informatik
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik 4
(Verteilte Systeme
und Betriebssysteme)



Performance Analysis of J2EE-Based Software Architectures and Application Servers

**(Performanzanalyse von Enterprise-Java-Software-
Architekturen und Applikations-Servern)**

Studienarbeit im Fach Informatik

vorgelegt von

Felix Eichhorn

geb. am 16.12.1976 in Crailsheim

Angefertigt am

Institut für Informatik

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: ***Prof. Dr. F. Hofmann***

Dr.-Ing. J. Kleinöder

Dipl.-Inf. B. Schnitzer

Dipl.-Sozw. C. Wehrfritz, 100world

Dipl.-Inform. (FH) U. Winter, 100world

Beginn der Arbeit: 02.01.2001

Abgabe der Arbeit: 24.04.2001

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 24.04.2001 _____

Diese Arbeit ist mit dem Ziel der Veröffentlichung in einer Fachzeitschrift entstanden. Es wurde deshalb besonderer Wert auf eine kurze Darstellung gelegt.

Abstract

The Java 2 Enterprise Edition (J2EE) provides a framework for the development of portable and distributed applications by combining the advantages of the Java platform with a component-based application model.

A J2EE application can be designed in many different ways. At the same time, as J2EE is just a specification, the J2EE implementations of different vendors, called J2EE application servers, might differ considerably in terms of reliability, performance and cost.

This paper compares the performance of different J2EE software architectures and application servers thus showing the importance of early considerations about the software architecture and application server used for realizing a J2EE application.

Kurzfassung

Die Java 2 Enterprise Edition (J2EE) kombiniert die Vorteile der Java Plattform mit einem komponenten-basierten Anwendungsdesign. Dadurch wird ein umfassendes Framework für die Entwicklung von verteilten Anwendungen zur Verfügung gestellt.

Eine J2EE-Anwendung kann auf viele unterschiedliche Arten aufgebaut sein. Da J2EE nur eine Spezifikation darstellt, können sich gleichzeitig auch die J2EE-Implementierungen verschiedener Hersteller (J2EE-Applikations-Server) in wesentlichen Punkten wie Zuverlässigkeit, Performanz und Kosten deutlich unterscheiden.

Die vorliegende Arbeit vergleicht verschiedene J2EE-Software-Architekturen und Applikations-Server im Hinblick auf ihre Performanz. Dadurch wird die Notwendigkeit aufgezeigt, schon in einer frühen Phase des Anwendungsdesigns Überlegungen zu der verwendeten Software Architektur und dem eingesetzten Applikations-Server anzustellen.

Table of Contents

Abstract	i
Kurzfassung	i
Table of Contents	ii
List of Figures	iii
1 Introduction	1
1.1 Overview	1
1.2 Goal of this Paper	2
1.3 Outline	2
2 The Java 2 Enterprise Edition	3
2.1 J2EE APIs	3
2.1.1 Servlets	3
2.1.2 EJBs	3
2.1.3 JDBC	4
2.2 J2EE Application Servers	5
3 Test Design	6
3.1 Test Application Design	6
3.2 Test Setup	8
3.3 Problems and Limitations	9
4 Test Results by Application Server	10
4.1 Orion 1.4.7	10
4.2 Resin 1.2.3 + jBoss 2.0	11
4.3 BEA WebLogic 6.0	12
4.4 Resin 1.2.3 + BEA WebLogic 6.0	14
4.5 IBM Websphere 3.5	15
5 Test Results by Application Architecture	16
5.1 Servlets	16
5.2 Stateless Session Beans	17
5.3 Stateful Session Beans	18
5.4 CMP Entity Beans	19
5.5 BMP Entity Beans	20
6 Conclusions and Future Development	21
7 References	22

List of Figures

Fig. 1: Role of an application server in a three-tier application architecture	1
Fig. 2: Example scenario	2
Fig. 3: Comparison of different types of EJBs	4
Fig. 4: Application server comparison	5
Fig. 5: Design of test application 1 (servlet)	6
Fig. 6: Screenshot of test data displayed in the client's web browser	6
Fig. 7: Design of test application 2 and 3 (stateless and stateful session bean)	7
Fig. 8: Design of test application 4 and 5 (CMP and BMP entity bean)	7
Fig. 9: Test setup	8
Fig. 10: Orion 1.4.7 performance charts	11
Fig. 11: Resin 1.2.3 + jBoss 2.0 performance charts	12
Fig. 12: BEA WebLogic 6.0 performance charts	13
Fig. 13: Resin 1.2.3 + BEA WebLogic 6.0 performance charts	14
Fig. 14: IBM Websphere 3.5 performance charts	15
Fig. 15: Servlet performance charts	16
Fig. 16: Stateless session bean performance charts	17
Fig. 17: Stateful session bean performance charts	18
Fig. 18: CMP entity bean performance charts	19
Fig. 19: BMP entity bean performance charts	20

1 Introduction

1.1 Overview

An application server typically consists of a set of services or components that provide for the presentation, business and connectivity logic of a system [Hut99]. The appearance of application servers in the market dates back to 1994 [Fra98], but as there was no common standard for middleware components and services, portability and reuse of components was very limited [Sag00].

Since 1999, the Java 2 Platform, Enterprise Edition (J2EE), provides a unified standard for distributed, component-based Java applications. It clearly defines well-formed interfaces between the application server's object containers and the objects or components themselves, while maintaining main Java features such as platform independence, reusability and modularity. A J2EE platform implementation is called a Java application server, or Java web application server if application services are provided over HTTP. As most current application servers actually are Java web application servers, the term "application server" is used as a synonym for "Java web application server" throughout this paper.

Typically, an application server is associated with the middle tier of a three-tier application. A three-tier application consists of a front-end client (which could be a web browser, for example), the middle-tier application comprising the business logic, and a back-end application which is usually a database or transaction server. The application server provides one or more object containers, execution environments for business logic components, and supporting infrastructure such as messaging systems, transactional managers and database accessors.

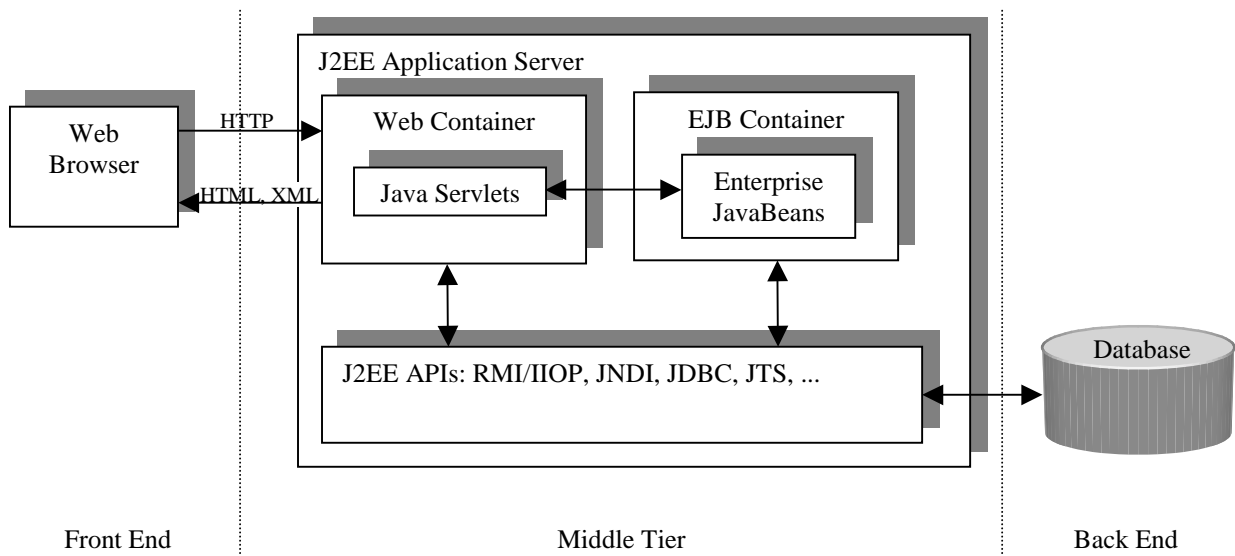


Fig. 1: Role of an application server in a three-tier application architecture

The two main object containers specified by J2EE are the web container, which hosts servlets and Java Server Pages (JSPs), and the Enterprise JavaBeans (EJBs) container for hosting EJB components. Servlets and JSPs are used for the simple and fast creation of dynamic web content, while EJB is the J2EE server-side component architecture. We will take a closer look at these components in chapter 2. The objects hosted by these containers have access to a set of J2EE Application Programming Interfaces (APIs), for example the Java Naming and Directory Interface (JNDI) for organizing and locating components in a distributed computing environment, and Java Database Connectivity (JDBC) which is a standard library for accessing databases.

An example usage of these J2EE components and interfaces is a banking application. In this example, all account data is stored in a back-end database. The business logic for money transfers and obtaining account information is provided by EJB components, which have access to the database via JDBC. The application server ensures transactional execution of business logic methods where necessary, for example for money transfers. Servlets and JSPs are used to dynamically create web content with the data and functionality provided by the EJB components. A bank customer then might use his web browser as client to interact with a servlet or a JSP via HTTP, for example to initiate a money transfer or to check his current account balance.

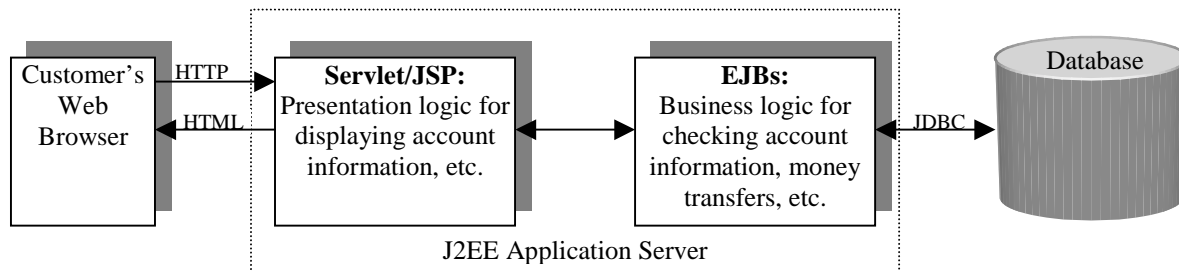


Fig. 2: Example scenario

The benefits of using an application server include reusable business logic components, better product focus as many important services are already provided by the application server, interoperability between different applications, and faster application development. However J2EE is just a specification and does not define any implementation details for an application server.

1.2 Goal of this Paper

At the moment there are more than 30 different application servers available, and they all differ in terms of reliability, speed and cost because they are implemented differently. Another concern are the many different ways of implementing a specific J2EE web application.

The goal of this paper is to compare some simple J2EE web application architectures, and to analyze the performance differences of these architectures when they are deployed on different application servers. The results will help to choose a suitable application architecture and application server when designing a J2EE application.

1.3 Outline

In *chapter 2*, an overview of the main J2EE components used in this paper, such as servlets and EJBs will be given, as well as an overview of the different application servers chosen for testing. *Chapter 3* describes the design of the applications implemented for performance testing, together with the hardware and software setup of the testing environment. This chapter also deals with the restrictions and problems we encountered during implementation and testing. The test results for each application server tested are presented in *chapter 4*. This chapter compares the performance of different application architectures on one application server. The same results are used in *chapter 5* to compare the test results of one specific application architecture on different application servers. *Chapter 6* summarizes the conclusions derived from the test results. It also provides a short outlook on the future development of J2EE and the application server market.

2 The Java 2 Enterprise Edition

2.1 J2EE APIs

At first, we have to take a closer look at the different components a typical J2EE application might consist of. As mentioned earlier, these are EJBs [Sub00,Dic99,Paw00], servlets [Sub00,Ber99] and JSPs. As JSPs are translated into servlets before execution, only servlets will be examined in this paper.

Servlets and EJBs might use other APIs such as JDBC for database access or JNDI for locating other components in a distributed environment. Of course there are many other APIs defined by the J2EE specification, but explaining them all in detail would go far beyond the scope of this work, so only those actually used for the test applications are introduced here.

2.1.1 Servlets

Today, most web servers use the Common Gateway Interface (CGI) to process user input and to serve dynamic web content. But performance, scalability and security are big problems when using Perl, which is one of the most common programming languages for CGI. Several web server vendors tried to solve these issues by providing proprietary APIs. The disadvantage of using these proprietary APIs is the fixation of your programs to one specific server vendor.

With the Java Servlet API, Sun tried to solve the issues of CGI and proprietary APIs with the help of the Java platform. Servlets provide a way to generate dynamic web content, are easy to develop, and have access to a set of other APIs, such as JavaMail and JDBC. Additionally, the Java security model makes it possible to implement a fine-grained access control, for instance only allowing access to a well-defined part of the file system.

Servlets are Java classes which handle HTTP client requests. For example, a servlet could process data posted over HTTPS using an HTML form, including purchase order or credit card data. Based on that data, a HTTP response is created and sent back to the client.

Servlets are executed in the web container of an application server. They may comprise some or all of the application's business logic, but often contain only the presentation logic and forward incoming requests to another component containing the application's business logic (for example an EJB).

2.1.2 EJBs

Enterprise JavaBeans are used to implement the business logic of an application. There are two types of EJBs in the EJB 1.1 specification: Session Beans and Entity Beans. Session beans implement business tasks, whereas entity beans represent business entities.

Session Beans. A Session Bean [Raj00] performs operations on behalf of a client, such as database access or performing calculations. It is relatively short-lived, since its lifetime is limited to that of the client.

There are two types of session beans. *Stateful Session Beans* keep state between method invocations. They are assigned to a client session and therefore must remain running after each method invocation. A client must connect with the same instance for each method call within the same session. Stateful session beans can be used for example to implement a shopping cart, which has to maintain its state during the shopping session of the client. The products already put in the cart then represent the state of the stateful session bean. There has to be one instance of a stateful session bean for every client.

Stateless Session Beans have no internal state and therefore may be destroyed after each method invocation or reused by another client. A client could also connect to a new instance every time, as the result of a method invocation does not depend on earlier method invocations. Normally stateless session beans will be used to service multiple clients for better performance. An example for the use of a stateless session bean is a currency converter that gets an amount of one currency as a method parameter and returns that amount in another currency. Multiple clients could call that method without interfering with each other.

Entity Beans. In contrast to the short-lived session beans entity beans represent data stored in a database in an object-oriented way and provide methods to act on that data. For example, a customer table in a relational database is represented by one entity bean per customer in that table. Therefore there is one entity bean per table row in this case.

Entity beans are long-lived as they exist as long as the data remains in the database. To be able to store an entity bean's state in or retrieve it from the database, the entity bean must either implement its own database access code (*bean managed persistence, BMP*) or delegate this task to its container (*container managed persistence, CMP*). Entity beans can be used by many clients simultaneously, and for performance improvements the container may cache or replicate entity beans, but this has to be done in a transaction safe and transparent way for the programmer.

Bean Type	Main Tasks	Characteristics
Stateless Session Bean	Implement business tasks, such as money transfer for a bank account	Do not maintain any state between client requests
Stateful Session Bean		Maintain state between request from one client
CMP Entity Bean	Represent data, such as the data associated with an bank account (owner, balance, ...) in an object oriented way	Persistence methods provided by the EJB container
BMP Entity Bean		Persistence methods have to be provided by the Bean developer

Fig. 3: Comparison of different types of EJBs

2.1.3 JDBC

JDBC is the standard Java API for accessing databases, and can be used by servlets and EJBs to implement database access functionality. In addition to the standard JDBC API, most application servers provide an implementation of the JDBC 2.0 Optional Package API. Some of the additional features in this package are support for connection pooling and distributed transactions. Connection pooling remarkably improves data access performance, as database connections are not opened and closed for every database access but can be reused for multiple database accesses and clients. Connections pools are managed by the application server.

Even with only this basic knowledge of the J2EE, it is easy to see that there are a lot of ways of implementing a specific J2EE web application. For the goal of comparing the performance of different J2EE software architectures, we implemented a simple web application in five different ways, one for each of the four EJB types mentioned above, and one without using EJBs at all. We will take a closer look at these applications in chapter 3.1. The following chapter will present the J2EE application servers chosen to test our applications on.

2.2 J2EE Application Servers

Normally it shouldn't be a problem to deploy any J2EE application on any J2EE compliant application server, but as J2EE is only a specification, each application server vendor implements its application server differently. Performance therefore is not only a question of application design but also depends considerably on the application server the application is deployed on. In this chapter we will take a closer look at the different application servers used for testing our applications.

We chose to compare some quite different application servers within the scope of this work. The two most commonly used application servers on the market today are BEA's WebLogic Server and IBM's Websphere [Ver00]. These are the most expensive products too, so a third application server, Evermind's Orion, is a relatively cheap one and of course there's an open source project too, called jBoss. As jBoss provides only an EJB container, Resin, an open source servlet engine was tested together with jBoss and the combination of Resin as servlet engine and BEA WebLogic as EJB container was examined too.

The following table lists the tested application servers together with the version tested, the supported J2EE containers and the approximate price.






Product Name and Version	Supported J2EE Containers	Price (approx.)
 BEA WebLogic Server 6.0	Servlets + EJBs	\$3000 - \$35000 per CPU
 IBM Websphere 3.5	Servlets + EJBs (1.0 only)	\$10000 per CPU (Adv. Edt.)
 Orion 1.4.7	Servlets + EJBs	\$1500 per Server
 jBoss 2.0	EJBs only	Free Open Source
 Resin 1.2.3	Servlets only	\$500 per Server Open Source

Fig. 4: Application server comparison

The test applications implemented to test the performance of these application servers are the topic of the following chapter.

3 Test Design

3.1 Test Application Design

One of the decisions to make before implementing a J2EE application is to see if the benefits provided by EJBs fit into your application, and if so, what types of EJBs to use. Of course, a large application might consist of several or all types of EJBs, but as we not only wanted to compare the performance of different application servers, but also to examine different application architectures, we chose to implement five basic test applications, one each for the four types of EJBs, and another one that doesn't use EJBs at all.

As we tried to keep these test applications as simple as possible they only provide access to some data stored in a database, which is something most application will need to do. All test applications use a servlet the client interacts with. When that servlet is called, it creates a random number and returns the record with that number as primary key from the database. More database details can be found in chapter 3.2. The database access can be realized either directly by the servlet or by using some type of EJB, but all database access takes advantage of connection pooling to improve performance. We now will take a closer look at each of the five test applications.

Servlet. The first possibility of implementing such an application is a servlet that directly accesses the database. As mentioned above, every time the servlet is called, it generates a random number and uses a (pooled) JDBC database connection to read the corresponding record from the database. It then displays that record in the client's web browser (test application 1). No EJB is needed for this test application.

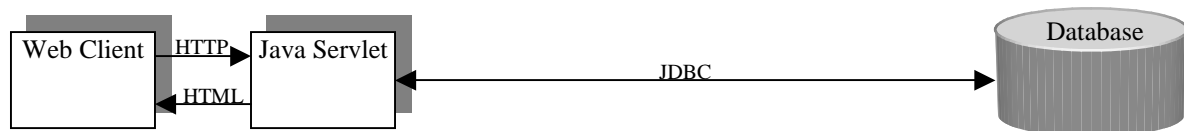


Fig. 5: Design of test application 1 (servlet)

Stateless Session Bean. When using a stateless session bean, all database access code is in that bean. The servlet gets a request from a web client, generates a random number and calls the corresponding method (for example `readData(int index)`) on the bean with the random number as a parameter. The stateless session bean then accesses the database as the servlet has done in test application 1 and returns the result to the servlet, which displays it in the client's web browser (test application 2).

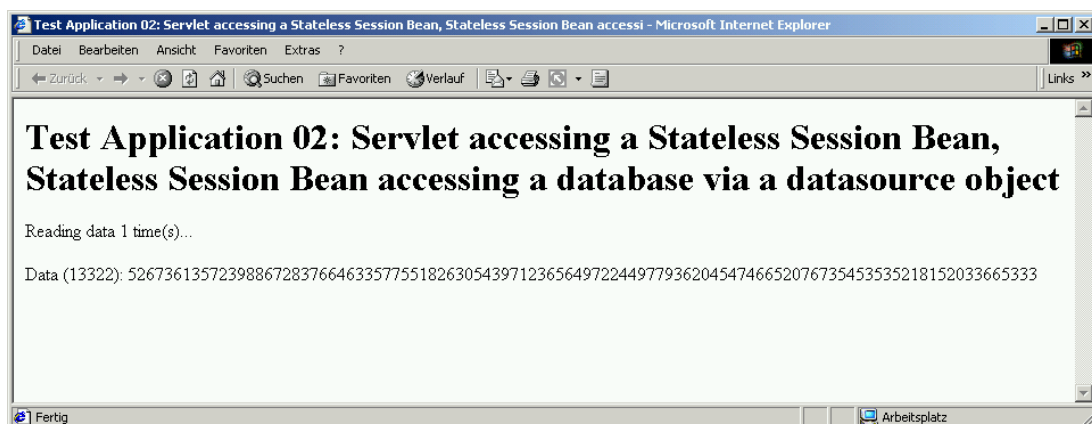


Fig. 6: Screenshot of test data displayed in the client's web browser

Stateful Session Bean. The state of the stateful session bean in our application is used to remember the record which was accessed last. In addition to providing a method to read a specific record from the database (readData(int index)), the stateful session bean provides another method (readNext()), which reads the record that directly follows the record accessed last from the database (test application 3).

When reading large record sets sequentially, with a stateless session bean the state has to be kept in the servlet and sent to the stateless session bean each time a new record is fetched. With the stateful session bean, only the first record has to be accessed directly and all other records can then be fetched by subsequent calls to the readNext() method. The state in the stateless session bean in our case is minimal (just one integer value), however for the application server there is a big difference between running one stateless session bean that all clients can use and running one stateful session bean for each client. So for performance comparisons, this minimal state is sufficient.

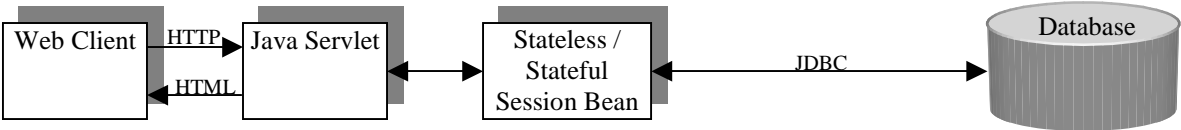


Fig. 7: Design of test application 2 and 3 (stateless and stateful session bean)

Entity Beans. Instead of directly accessing the database, it is also possible to use entity beans for an object-oriented view on the data stored in the database. To be able to look up a specific record, a so called finder method of the entity bean is called, such as findByPrimaryKey(). The resulting entity bean then represents the data of the corresponding record in the database. For the test applications a stateless session bean is used to handle entity beans.

CMP Entity Bean. In one case (test application 4) the entity bean is a CMP entity bean, so there's no need to write any database access code at all. Only the mapping between the database columns and the entity bean's fields has to be provided, the rest will be done by the application server.

BMP Entity Bean. With BMP entity beans, the code for reading the entity bean's fields from the database, writing them back when necessary, finding specific records in the database, creating new records or deleting records has to be written by the bean developer (test application 5).

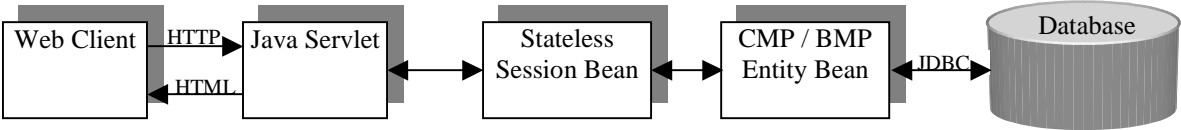


Fig. 8: Design of test application 4 and 5 (CMP and BMP entity bean)

This chapter should have provided a basic idea of what the test applications do and what the differences between the 5 application architectures are. For implementation details, please refer to the source code, which is available from the author. The next chapter deals with the hardware and software used to run the application servers and test applications and to measure their performance.

3.2 Test Setup

The hardware used for running the application servers were two Intel PIII machines with 733 MHz and 256MB of RAM. They were running SuSE Linux 7.1, Kernel 2.4.0.

The database was an Oracle 8.1.6 Enterprise Edition database running on a Sun Ultra-4 machine with 2GB of RAM. The test data in the database consisted of 100000 records, each with an integer value as the primary key and a string of length 100 as a data field. For every client HTTP request to one of our test applications, one random record was read from the database, sent back to the client with the HTTP response, and displayed in the clients web browser. The HTTP responses were about 490 bytes in length.

To simulate different numbers of clients (10, 100, 250, 500, 750) accessing the test applications, an Intel PIII computer with 128MB of RAM, Windows 2000 and the Microsoft Web Application Stress Tool version 1.1.293.1 were used. This program generates HTTP requests for every simulated client and collects statistical data about these requests. The data we used in this paper were the throughput (successful requests per second, RPS), the response time (median value of time from sending the HTTP request until the last byte of the HTTP response was received, in milliseconds), and the error rate (the percentage of requests that returned an error, for example because no database connection was available within a specific amount of time).

We chose the median response time value as both successful and unsuccessful request are used by the Microsoft Web Application Stress Tool to calculate response time values. As unsuccessful requests may take considerably more or less time, the median value gives a more accurate estimation of the average response time of a successful request than the arithmetic mean.

All tests ran 7 minutes: 2 minutes warm-up time during which no measurement took place, and 5 minutes test time.

All computers were connected with a switched 100Mbps network.

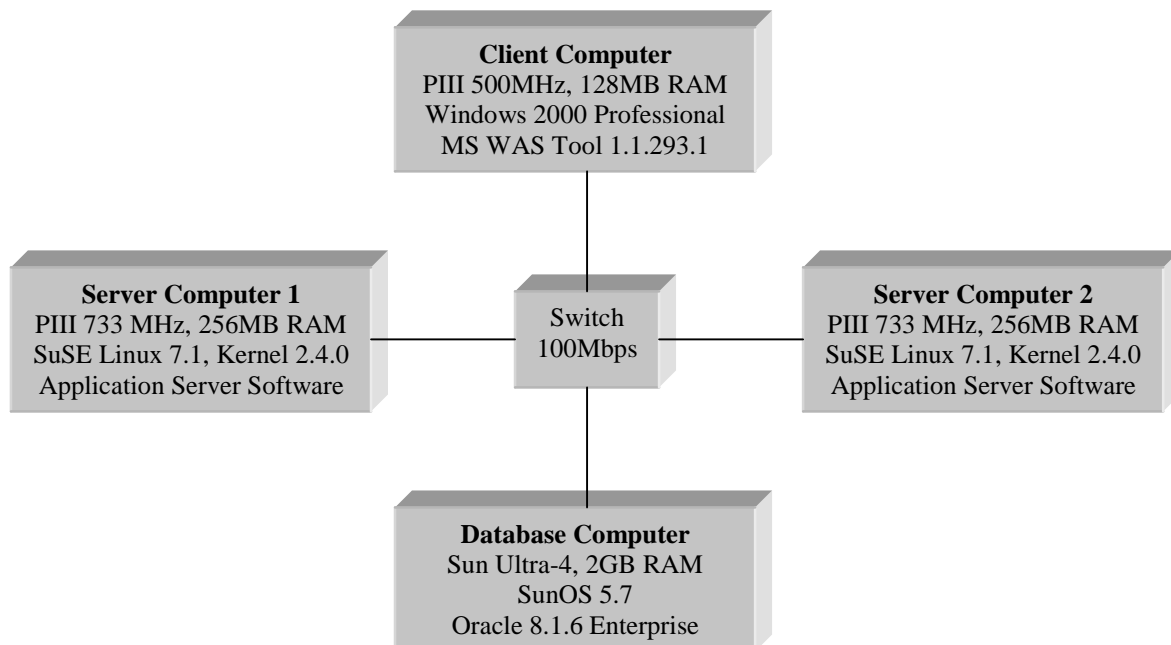


Fig. 9: Test setup

Every test application was tested on every application server, once with the web and EJB container running together on the same machine (server 1), and another time with only the EJB container running on server 1 and with the servlet container running on server 2.

All application servers were configured to provide a connection pool for database connections. The size of the connection pool was set to 10. For all other configuration options the default settings were used. More on configuration options can be found in chapter 3.3.

The transaction attributes of all EJBs were set to “supports”, so transactional execution would have been possible, but as a transaction never actually was initiated, all execution took place in a non transactional context.

The Java implementation used was Sun’s JDK 1.3.1 Beta 1 for Linux, only IBM Websphere used IBM’s JDK 1.2.2 because it is shipped and installed with IBM Websphere.

3.3 Problems and Limitations

One of the main problems was that there is no such thing as a standard web application. Every application has totally different goals and needs. So we tried to focus on one aspect most web applications share, that is database access, and measured the data access performance for different application architectures and application servers.

Other aspects, like the speed of complex calculations, mainly depend on the hardware and algorithms used, not on the specific application server. For reliability or security tests, a totally different test architecture is necessary. And of course there are a lot of other points worth examining, which could be the topic of future papers, but go beyond the scope of this work.

It was also impossible to test every application server on the market, so the application servers chosen for this paper were the two main competitors on the market today, BEA WebLogic and IBM Websphere, and three cheaper or open source servers that could be attractive especially for smaller companies.

With the test applications, it would also have been possible to measure the performance when inserting, updating or deleting data in the database. But when some of these tests were run for the first time, we noticed that not the application server, but the database was the bottleneck here. For these tests, the server that contains the database needs to be optimized to handle many concurrent write operations. However such a server was not available, so these tests were dropped.

As all application servers have a lot of configuration and fine-tuning options, which are often completely different between two vendors, we decided to use just the standard configuration for testing. In the first place, this helps to keep the different application servers comparable, but within the limits of this work it additionally would have taken too much time to examine all these options in detail. In a real world environment all these options have to be examined and adjusted especially for a specific application and client behavior.

Similarly it is possible to optimize an application for a specific application server by using non-standard extensions or by just analyzing the application server’s behavior in a specific situation and to adapt the application to that behavior. However that behavior could be totally different on another application server. As we wanted to test the same applications on different application servers, they were not optimized for a specific one (with the exception of a special servlet version for Resin, see chapter 4.2).

We also encountered some bugs in Sun’s JDK for Linux, which sometimes lead to application server crashes. All bugs were already known to Sun and will hopefully be fixed soon.

In the following chapter the compiled results of our tests can be found, together with explanations on some points of particular interest.

4 Test Results by Application Server

In this chapter, the test results for every application server will be presented. There is a diagram for the throughput (successful request per second, RPS), the response time (in ms) and the error rate (in percent) for every application server. Diagrams on the left hand side show the results for the single server setup (web and EJB container running on one computer), the diagrams on the right hand side show the results for the dual server setup (web and EJB container running on two different computers). The diagrams for the dual server setup do not show the results for the servlet application (test application 1) as the EJB container is not involved in this test and therefore the results would be exactly the same as for the single server setup. The x-axis indicates the number of simulated clients. Every test was run for 10, 100, 250, 500 and 750 clients.

Normally you would expect the servlet without EJBs to perform best, as it directly accesses the database without having to communicate with EJBs first. On the other hand, the servlet won't be easy to reuse for other applications and also all database access code has to be written by the programmer, in contrast to CMP entity beans.

Stateless session beans should perform better than stateful session beans as they may be used for multiple clients. As all entity beans are used only together with a stateless session bean, entity bean performance can be expected to be lower than the performance for the stateless session bean application.

Following the diagrams there will be a short discussion of relevant results and particularly results differing from the expected values will be looked at. A comparison between different application servers for one application architecture follows in chapter 5.

4.1 Orion 1.4.7

The most interesting result for the Orion application server was the high performance of CMP entity beans. After some testing we found out that Orion does not limit the cache size for CMP entity beans to a specific number of beans as all other application servers do. Our database consisted of 100000 records of 100 bytes each, therefore it was possible for Orion to cache the entire database, so after some warm-up time there is little to no database access necessary anymore. As it is not possible to limit the cache size for Orion manually, the results presented here apply to a filled cache only. In real-world applications, the results will depend significantly on the size of the database. For small databases Orion's caching mechanism will lead to high performance as in our test case. For large databases however, caching of the entire database won't be possible so that the results will be significantly lower.

While trying to test Orion with more than 750 users, we found out that Orion creates a new thread for every incoming client request. Because of a bug in Sun's JDK for Linux (which Sun attributes to a bug in the Linux kernel), it is not possible to create a large number of threads (about 800 in our case) under Linux. So Orion was constantly crashing when load tested with more than 750 users. This problem doesn't arise when running Orion under Windows, but might become a problem for heavy traffic sites using Orion under Linux.

Another problem that arises due to Orion's excessive use of threads is the constantly decreasing throughput. Most other application servers (for example BEA WebLogic, Resin) use some kind of thread-pooling, allowing only a limited number of threads to execute at a given time. Besides circumventing the JDK bug mentioned above, this helps to keep throughput values relatively constant, even for a high number of concurrent users, while only the response time values are getting worse.

As error rates were not exceeding 0.2 % for the Orion application server, there's not much to see in the error rate diagrams.

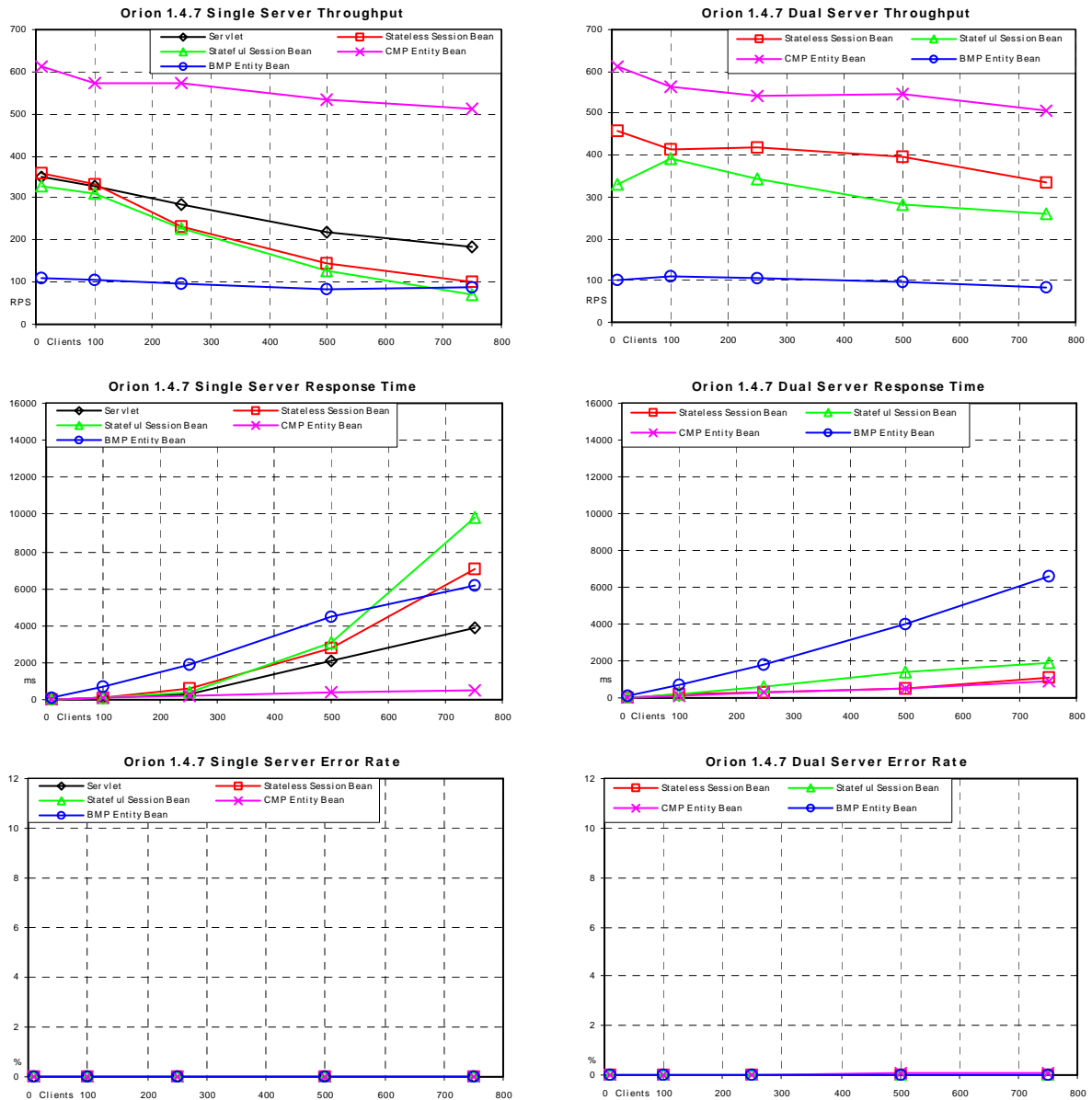


Fig. 10: Orion 1.4.7 performance charts

4.2 Resin 1.2.3 + jBoss 2.0

When testing servlet performance with Resin as the servlet container, Resin also has to provide the connection pool for database connections. As in all tests, the size of the connection pool was set up to a maximum of 10 connections. But if there are too many concurrent requests and Resin doesn't have any free connections in the connection pool, Resin opens up new database connections anyway, so the total number of connections will exceed the configured maximum size of 10. If the database is not able or just not set up to handle a very large number of concurrent connections, the performance decreases while the number of errors increases. This is what happened here, so with only 10 clients using the servlet application concurrently, the throughput and response time values are very good. But as soon as there are too many clients, and therefore too many concurrent database

connections, the throughput values drop, the response time increases up to 25 seconds and the error rate up to 11.8 percent.

Therefore we wrote a special Resin version of the test application that ensures that the number of database connection used does not exceed 10. The results for this special version are much better, as you can see in the diagrams, but on the other hand you need to write additional code you don't need for any other application server.

With jBoss as EJB container and connection pool provider, the throughput and response time for 750 users is slightly better than throughput and response time for 500 users. But for these numbers of users, the error rate is significantly higher too.

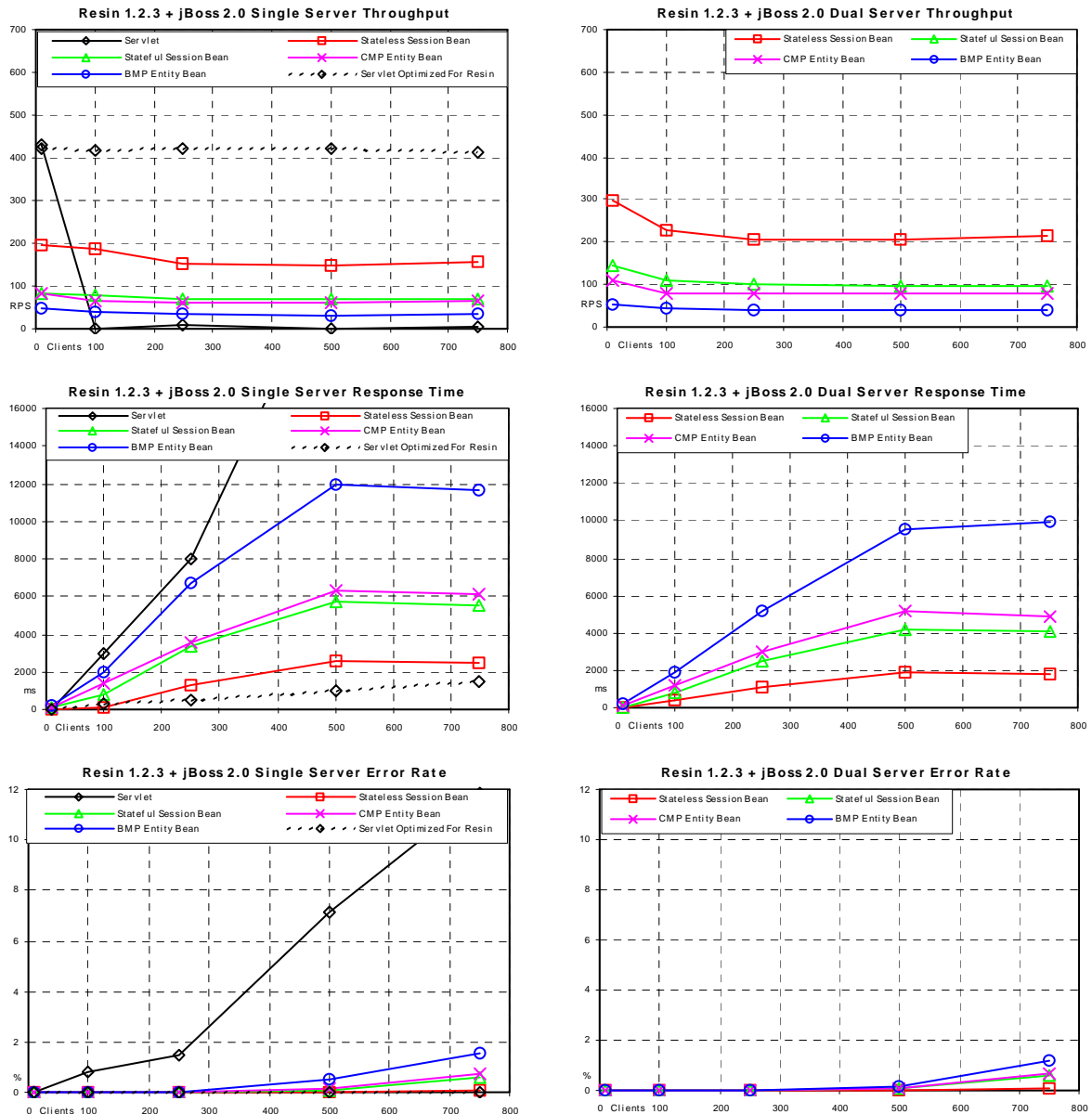


Fig. 11: Resin 1.2.3 + jBoss 2.0 performance charts

4.3 BEA WebLogic 6.0

For the BEA WebLogic application server the throughput and error rates stay on approximately the same level even for a growing number of users, only the response time values grow linearly.

The relatively large error rate can be explained with the database connection handling of BEA WebLogic. When there's no free connection for an incoming request in the connection pool, BEA WebLogic will wait a certain amount of time if a connection becomes available, and if there's still no connection available after that time, an error will be returned. Normally the servlet or the EJB that requires the connection would have to catch that error and try again to get a connection.

As some individual tests with an optimized servlet showed it is relatively easy to prevent these errors while the throughput values remain at nearly the same level. But as we didn't want to optimize our applications for a specific application server, and as other application servers handle that problem in other ways, we get these relatively high error rates here for the standard test applications without optimization.

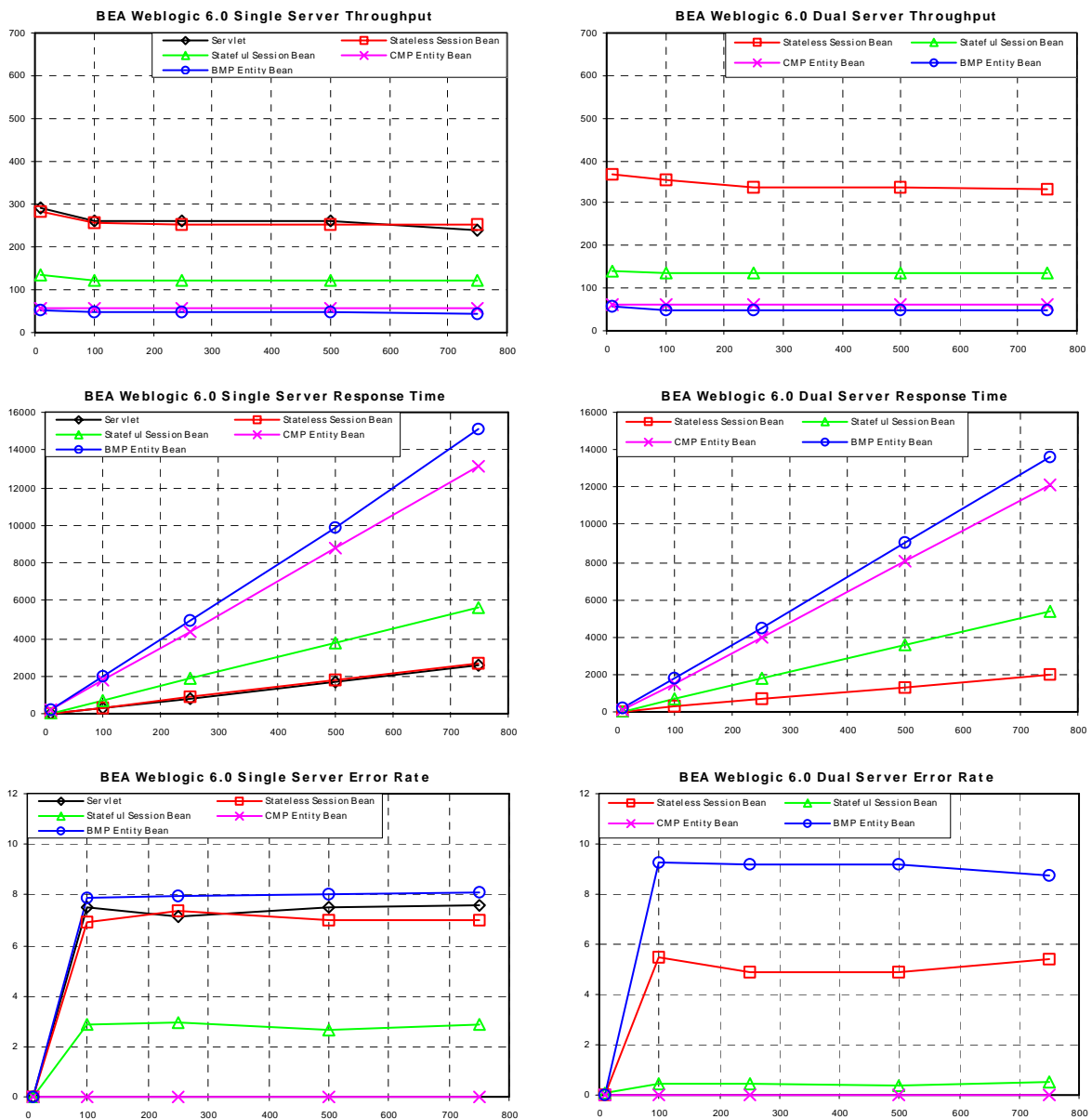


Fig. 12: BEA WebLogic 6.0 performance charts

When using CMP entity beans there's only a very small percentage of errors. That is because for CMP entity beans the container has to provide all database access code and therefore is responsible for preventing these errors, too.

4.4 Resin 1.2.3 + BEA WebLogic 6.0

It is also possible to combine the servlet engine of Resin with the EJB container of BEA WebLogic. As mentioned before, the relatively high error rates (up to 11.2 percent) stem from the way BEA WebLogic handles the database connection pool and returns errors if there's no connection available within a specific amount of time.

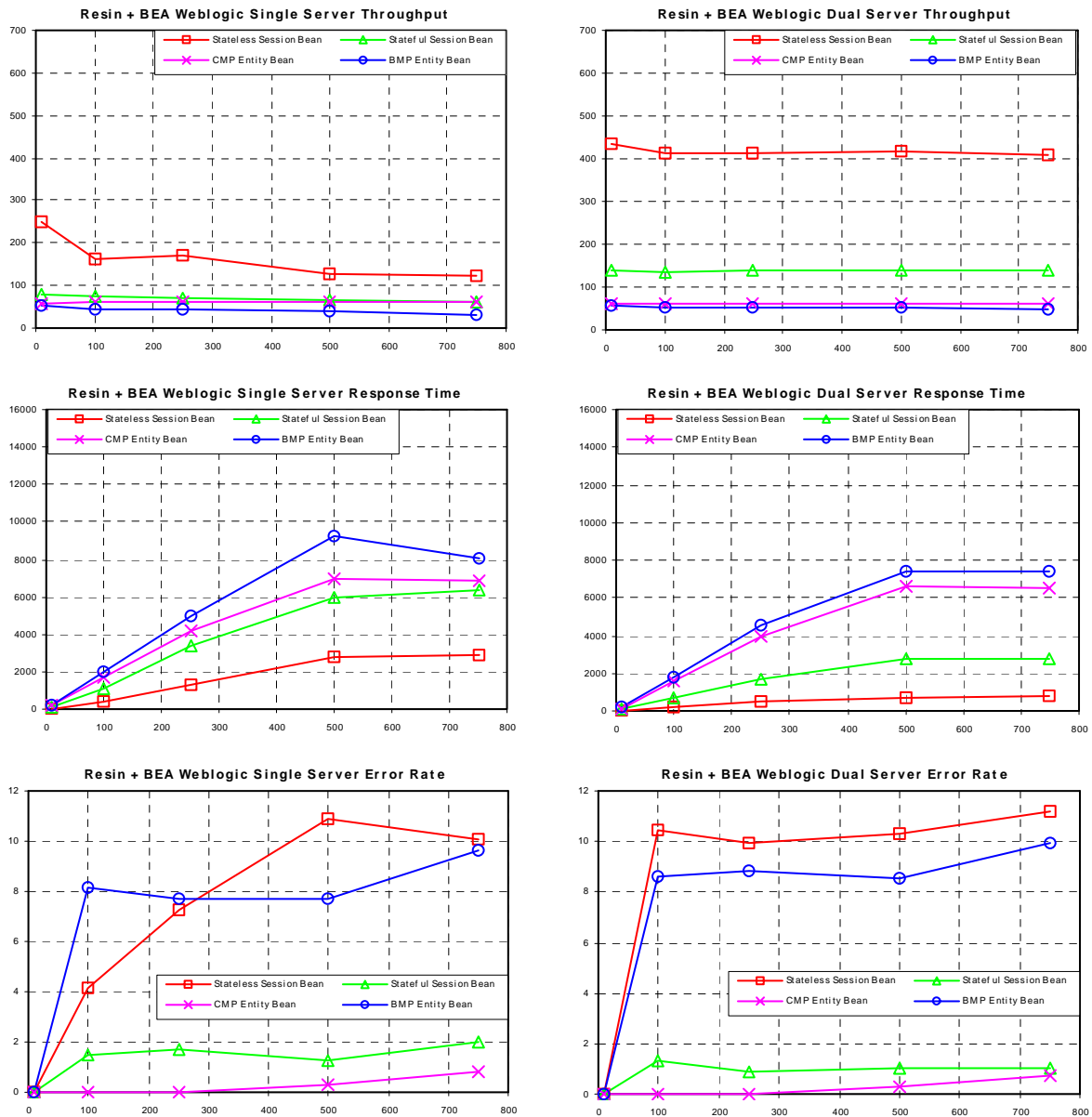


Fig. 13: Resin 1.2.3 + BEA WebLogic 6.0 performance charts

When using Resin together with BEA WebLogic, it is important that Resin is able to find the WebLogic classes to communicate with the BEA WebLogic Server. During our tests we found out that the way you make the WebLogic classes available to Resin plays an important role for the performance of the application. One possibility is to configure the (local) path to the WebLogic classes in the Resin configuration file, together with the application configuration. This configuration works without problems, but is approximately 50% slower than the second configuration, for which the results are shown here. For this configuration we just copied the WebLogic class files over to the Resin lib directory. We asked on the Resin mailing list what the reason for this behavior might be but didn't get an answer so far.

4.5 IBM Websphere 3.5

For IBM Websphere 3.5, only servlet and stateless session bean data is shown as all other application architectures were too slow in performance (only 2 requests per second for stateful session beans, 0,5 request per second for entity beans). The main reason for this bad performance seems to be the amount of memory we used in our test computers. IBM recommends at least 1GB of physical memory for running Websphere, whereas our computers only had 256MB. Another disadvantage was that IBM doesn't provide any service packs for the Linux operating system. These service packs, which according to IBM among other things improve performance, are only available for AIX, Solaris and Windows 2000/NT. Already four service packs are available from IBM.

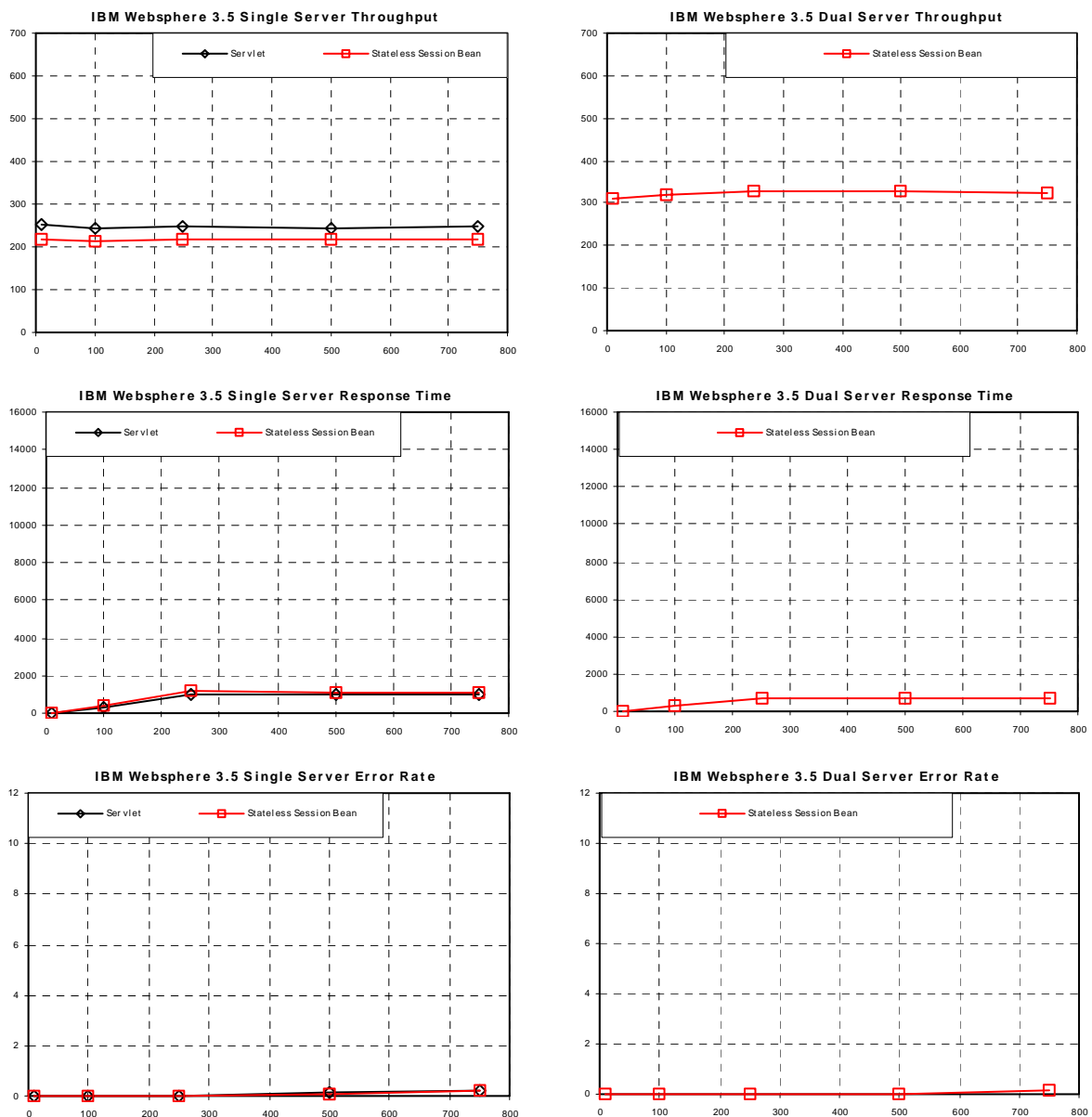


Fig. 14: IBM Websphere 3.5 performance charts

The following chapter will use the same results to compare the performance for one application architecture on different application servers.

5 Test Results by Application Architecture

In this chapter, all test results will be compared by application architecture. With the help of the diagrams it is easy to compare the performance of different application servers for a specific application architecture. Only throughput and response time are examined in this chapter, for error rates please refer to the preceding chapter.

5.1 Servlets

Servlets only run in the web container of an application server, therefore there is no data for the dual server configuration in this case. For Resin there are two result sets again, one for the standard test application which was run on all application servers, and one that was optimized for Resin database access behavior. More details on that topic can be found in chapter 4.2.

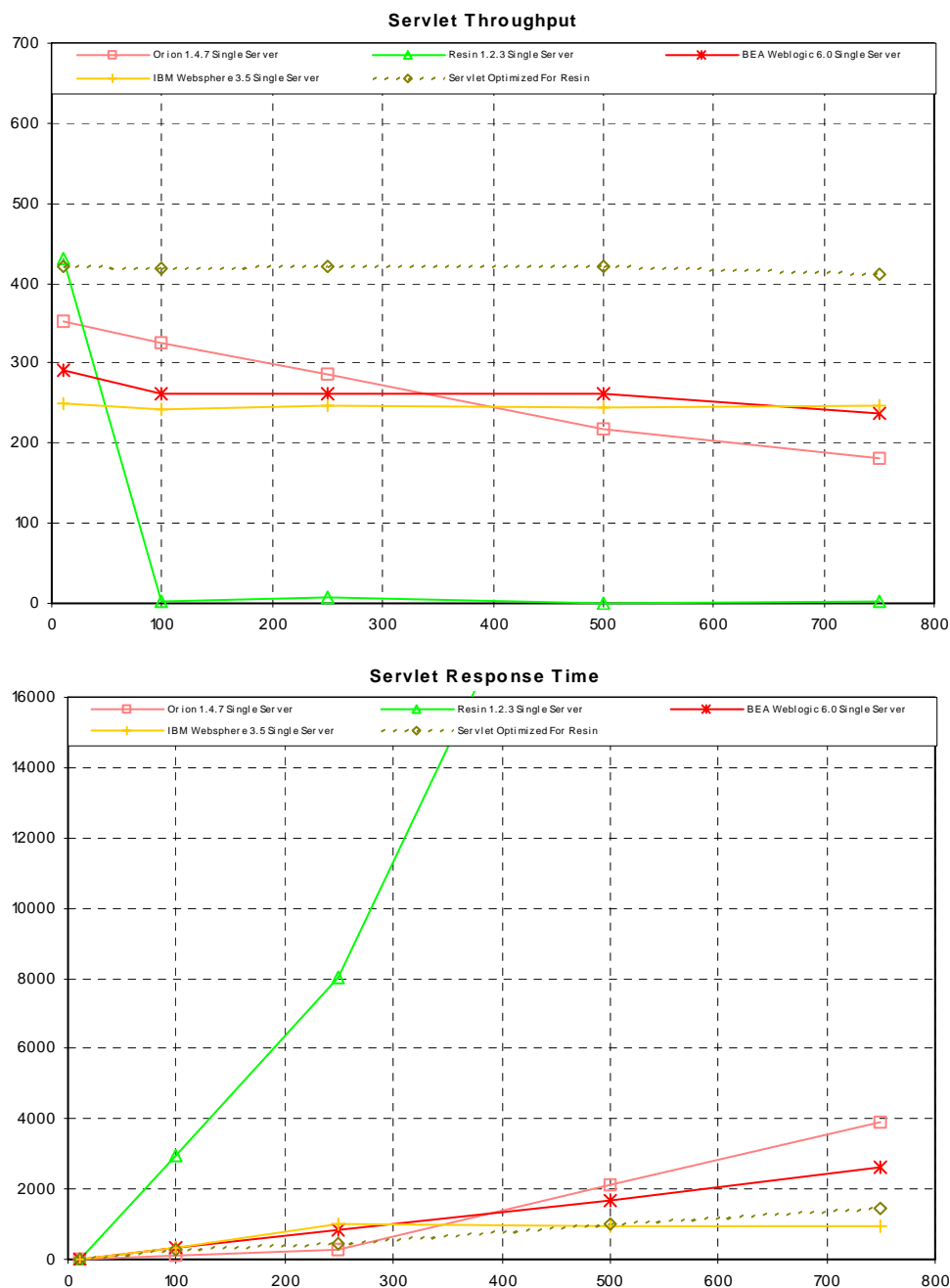


Fig. 15: Servlet performance charts

5.2 Stateless Session Beans

For a large number of concurrent users, the combination of Resin and BEA WebLogic provides very good throughput and response time values. For a small number of users Orion is marginally faster, but Orion's performance decreases continuously with an increasing number of clients.

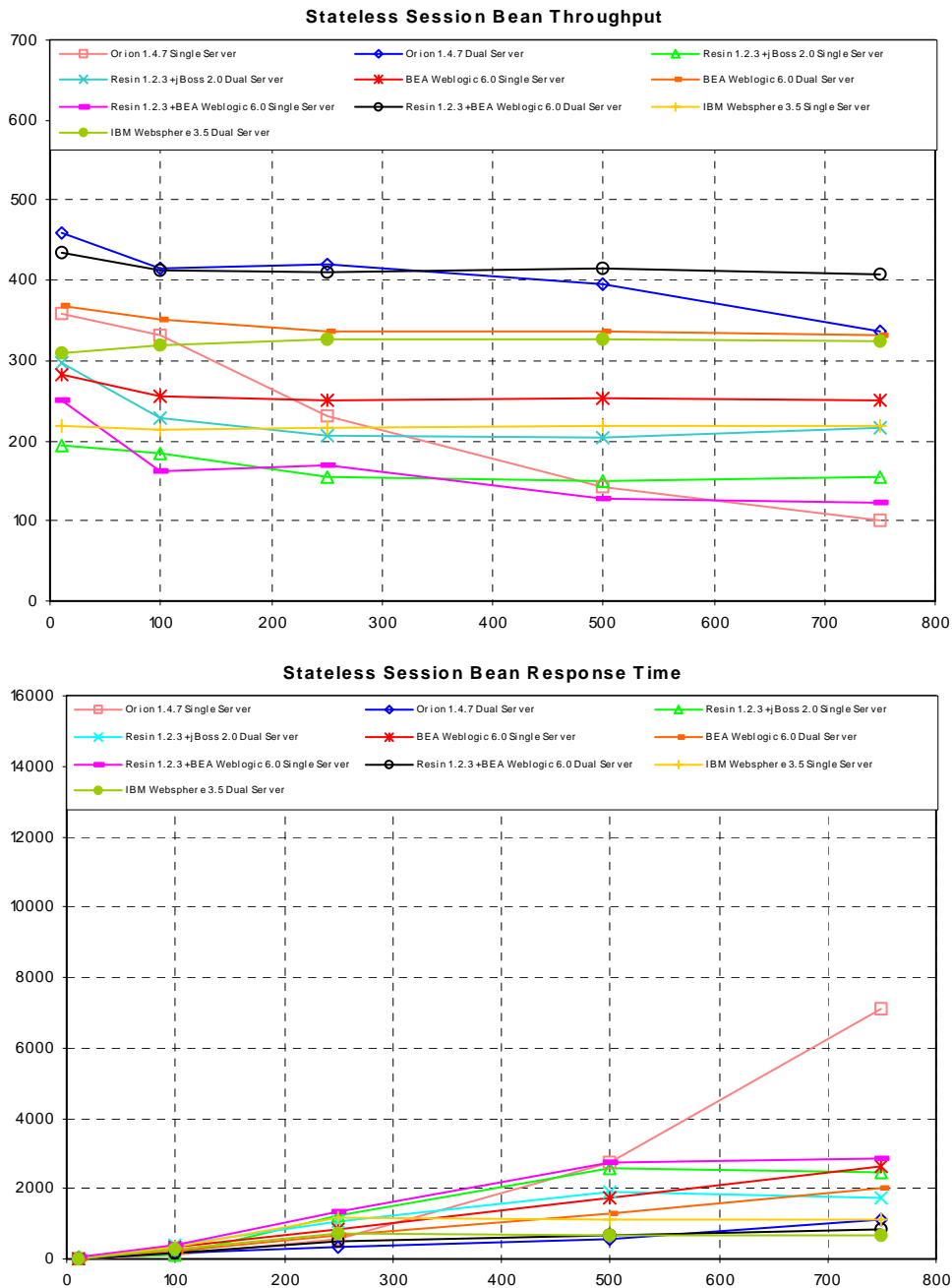


Fig. 16: Stateless session bean performance charts

This is visible even more clearly when looking at the results for the Orion single server configuration. Here the throughput drops from about 350 request per second for 10 users to about 100 requests per second for 750 users. One of the reasons for this decreasing performance is Orion's usage of threads, which is explained in chapter 4.1 in more detail.

The performance for Resin and BEA WebLogic stays at nearly the same level, even for a large number of users.

5.3 Stateful Session Beans

Stateful session beans in most cases are much slower than stateless session beans, as they have to maintain state for a single client and thus can't be reused for multiple clients.

For stateful session beans, the Orion application server in dual server configuration performs best even for a high number of users. But here too it is plainly visible that the performance is decreasing steadily, whereas for all other application servers the performance decreases only marginally from 10 to 100 users and then remains at nearly the same level.

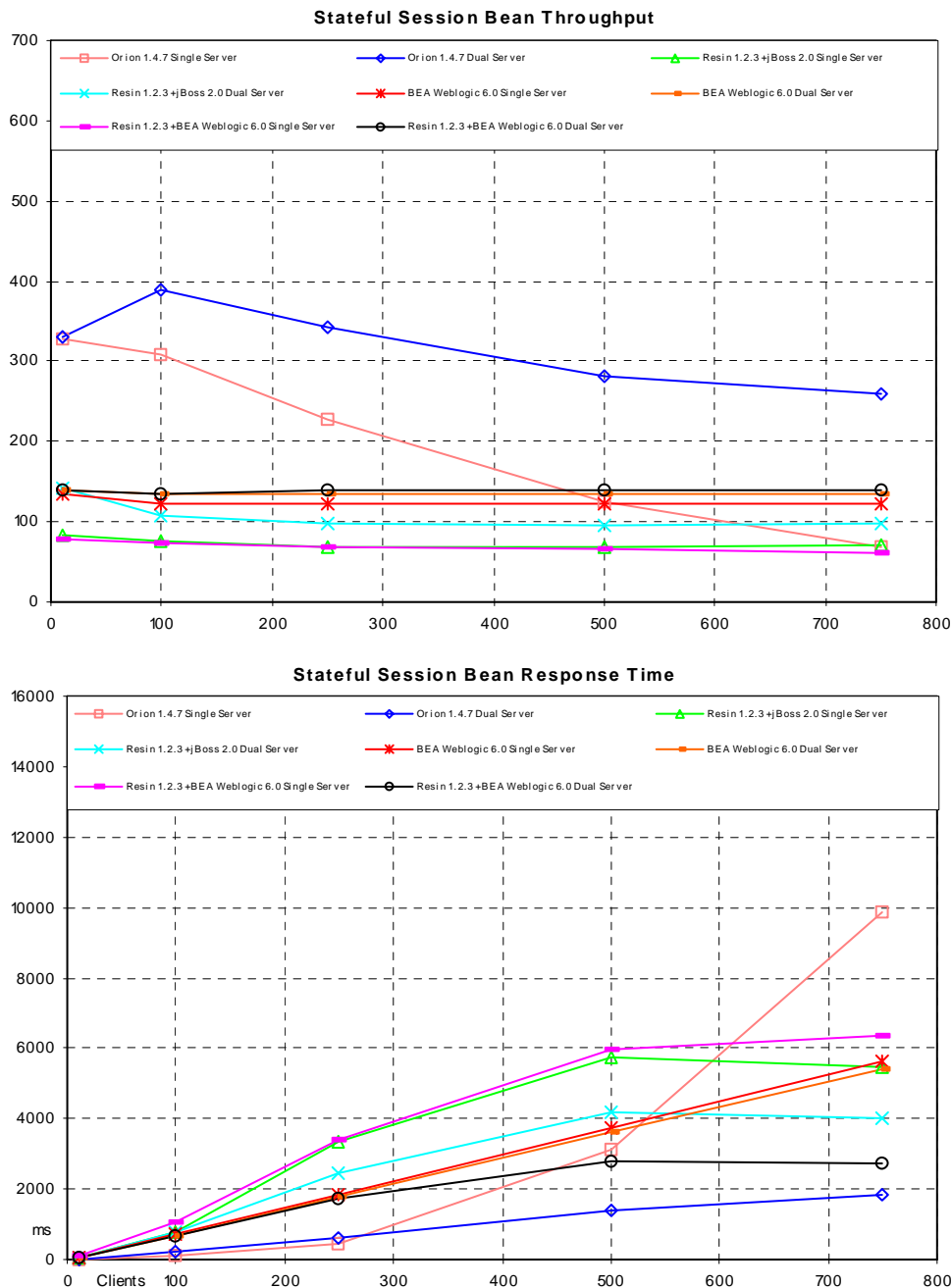


Fig. 17: Stateful session bean performance charts

No IBM Websphere results for stateful session beans, CMP entity beans and BMP entity beans are shown due to the bad performance of IBM Websphere for these application architectures. More explanations on IBM Websphere performance can be found in chapter 4.5.

5.4 CMP Entity Beans

The throughput values for container managed persistence entity beans are on a similar level for all application servers with the exception of Orion. Orion uses a larger cache and therefore was able to hold all the accessed data in memory. For more details on that topic please refer to chapter 4.1.

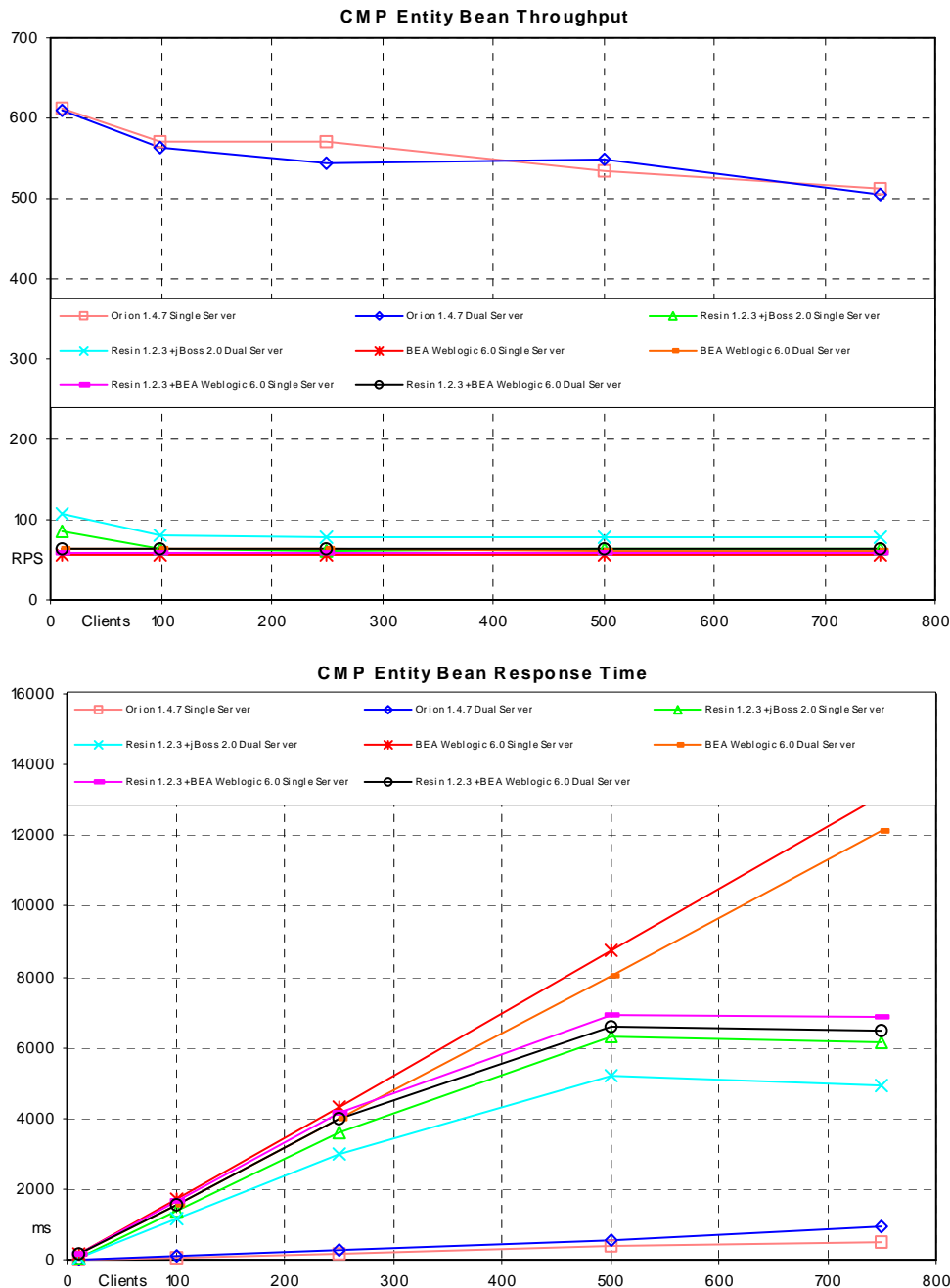


Fig. 18: CMP entity bean performance charts

Differences between the application servers can be recognized when looking at the response time. For BEA WebLogic, the response time increases linearly, whereas for Resin + jBoss and Resin + BEA WebLogic the response time values start to decrease again from about 500 concurrent users. The reason for this behavior is the increasing error rate (see chapter 4.2 and chapter 4.4). Errors usually don't take as much time to be returned as a successful request, so the response time values start to decrease with an increasing error rate.

5.5 BMP Entity Beans

The performance for BMP entity beans in most cases is worse than the performance for CMP entity beans. As CMP entity beans are also easier to develop, they should be used whenever possible. BMP entity beans however provide an opportunity to take advantage of the entity bean programming model where the use of CMP entity beans is not possible, for example when working with complex database schemas.

The response time values for Orion and BEA WebLogic again are increasing linearly, whereas for Resin + jBoss and Resin + BEA WebLogic the values are decreasing slightly between 500 and 750 concurrent users due to an increasing error rate.

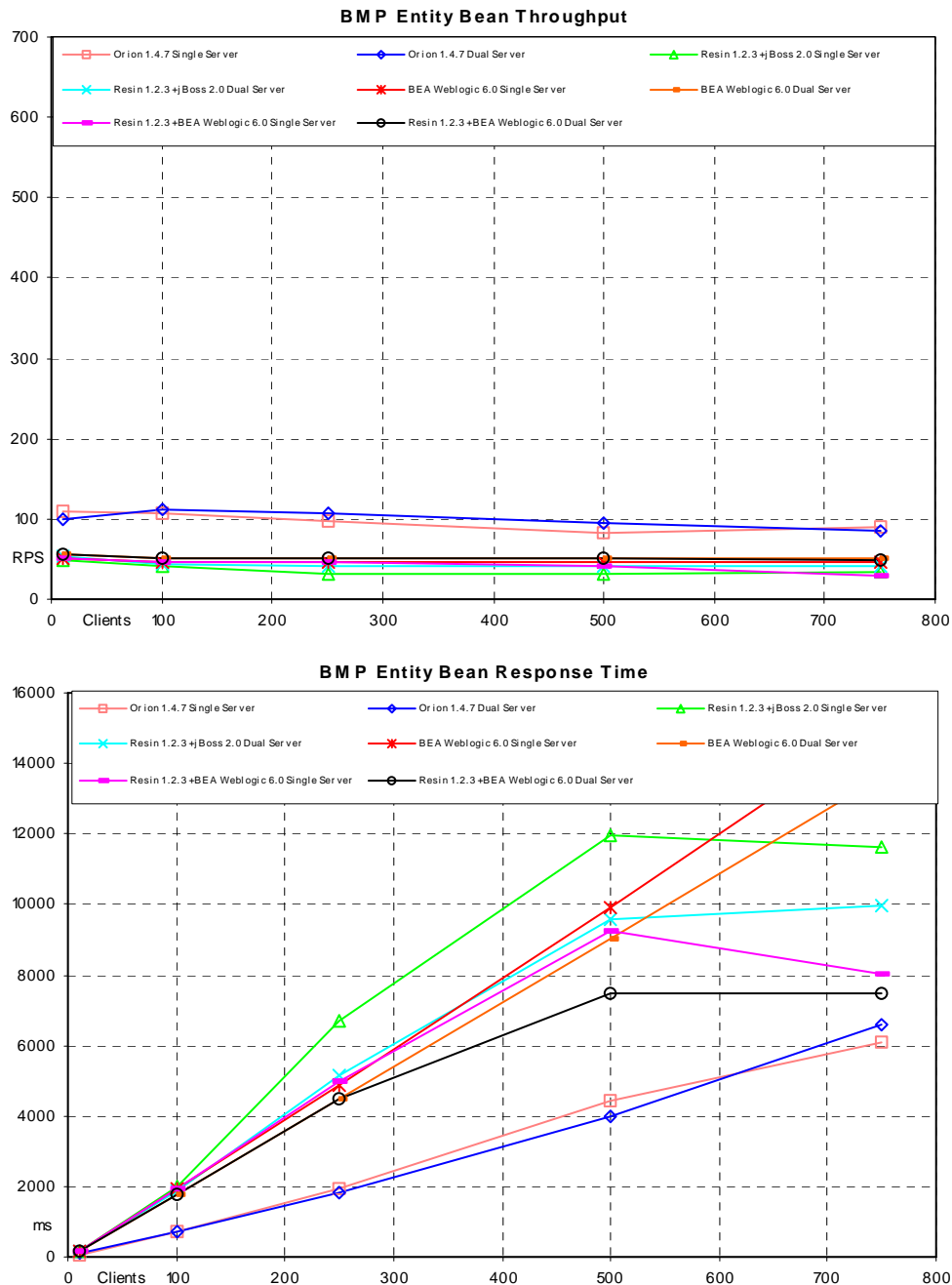


Fig. 19: BMP entity bean performance charts

The following chapter summarizes the results of this paper and takes a look at the future development of the Java 2 Enterprise Edition and the application server market.

6 Conclusions and Future Development

There are two main conclusions drawn from the test results. First, it is important to take performance requirements into account early when thinking about the design and architecture of a J2EE application. The best architecture is useless if your application can't produce the performance you need, and our tests showed that the performance may differ significantly between the application architectures. On the other hand, when performance is not the decisive factor, using for example CMP entity beans renders own database access code unnecessary and thereby helps to prevent bugs and save development time.

The other point worth paying attention to in an early phase of application development is the application server to use. Our tests showed that it is not possible to write an application that runs with optimum performance and least error rate on every application server. One of the main goals of the J2EE specification was the portability of components between different application server vendors. But had we taken into account application server specific behavior (for example when handling database connections), and optimized our applications for that behavior, higher performance and a lower error rate could have been achieved. The same holds for application server specific configuration options and extensions to the J2EE specification. More on the topic of EJB portability can be found in [DRS99].

Furthermore it is important to keep an eye on the fast growing and rapidly changing application server market and the future development of the J2EE specification. At the time of writing, the EJB 2.0 specification is nearly completed. It introduces asynchronous messaging through the use of a new type of EJB, the message driven bean. It also greatly improves the possibilities for working with CMP EJBs for example through the introduction of a standard EJB query language. Some of the application servers, such as BEA WebLogic, already support all or part of the new specification.

Nearly every week updates are released to provide bug fixes and performance improvements, so in the meantime BEA released the BEA WebLogic Server 6.1, jBoss 2.2 is available, IBM finished a first version of Websphere 4.0 for z/OS and OS/390, and the latest version of Resin is 1.2.5. Some of the problems we encountered during testing may already have been fixed by these updates, so keeping your application server installation up-to-date is important too.

Altogether, when looking for a low price application server, Resin in combination with jBoss is a good recommendation. Both provide good performance and, as they are open source, bugs possibly can be fixed without having to wait for the next service pack to be available. This could in my opinion become a problem with Orion, as it is developed by just two people at the moment, and bug fixes often take some time to be released. When scalability and high performance are more important than the price, the combination of Resin and BEA WebLogic is a good solution. But whichever application server you choose, when designing your application architecture it is important to take application server specific behavior into account early, as well as the performance requirements of your application.

7 References

- [Ber99] Bergsten, H.: *Java Servlets - An Introduction*.
http://www.webdevelopersjournal.com/articles/intro_to_servlets.html,
Web Developer's Journal, 1999.
- [Dic99] Dick, K.: *Introduction to Enterprise JavaBeans*.
http://www.middle-tier.com/EJBReport/EJBIntro_main.html,
MiddleTier.com, 1999.
- [DRS99] Dorda, S., Robert, J., Seacord, R.: *Theory and Practice of Enterprise JavaBean Portability*.
<http://www.sei.cmu.edu/pub/documents/99.reports/pdf/99tn005.pdf>,
Carnegie Mellon Software Engineering Institute, 1999.
- [Fra98] Francis, B.: *Rewriting History*.
<http://www.zdnet.com/eweek/news/0921/21appser.html>,
PC Week Online, 1998.
- [Hut99] Hutcheson, D.: *All-purpose definition of an enterprise application server*.
<http://www.enterprisedev.com/upload/free/features/entdev/1999/03mar99/app0399/app0399-1.asp>,
Enterprise Development, 1999.
- [Paw00] Pawlan, M.: *Using Session & Entity Beans To Write A Multitiered App*.
<http://java.sun.com/products/ejb/articles/multitier.html>,
Sun Microsystems, 2000.
- [Raj00] Raj, G. S.: *EJB Session Beans*.
<http://www.idevresource.com/java/library/articles/ejb-session.asp>,
IDevResource.com, 2000.
- [Sag00] Sagar, A.: *Anatomy of a Java Application Server*.
<http://www.sys-con.com/java/archives/0504/sagar/>,
Sys-Con Publications, Inc., New York, 2000.
- [Sub00] Subrahmanyam, A. et al.: *Professional Java Server Programming J2EE Edition*.
Wrox Press Ltd., Birmingham 2000.
- [Ver00] Verton, D.: *BEA, IBM Square Off In App Server Market*.
http://www.computerworld.com/cwi/story/0,1199,NAV47_STO52117,00.html,
Computerworld, 2000.