

# Designing a Test Suite for Empirically-based Middleware Performance Prediction

Yan Liu<sup>1</sup>, Ian Gorton<sup>1,2</sup>, Anna Liu<sup>2</sup>, Ning Jiang<sup>1</sup>, Shiping Chen<sup>2</sup>

<sup>1</sup>Basser Department of Computer Science  
University of Sydney

Sydney 2006, NSW, Australia

<sup>2</sup>CSIRO CMIS

Building E6b, Macquarie University, North Ryde  
Sydney 2113, NSW, Australia

jennyliu, jjiang@cs.usyd.edu.au

ian.gorton, anna.liu, shiping.chen@cmis.csiro.au

## Abstract

One of the major problems in building large-scale enterprise systems is anticipating the performance of the eventual solution before it has been built. This problem is especially germane to modern Internet-based e-business applications, where failure to provide high performance and scalability can lead to application and business failure. The fundamental software engineering problem is compounded by many factors, including application diversity, architectural trade-offs and options, COTS component integration requirements, and differences in performance of various software and hardware infrastructures. In the ForeSight project, a practical solution to this problem, based on empirical testing is being investigated. The approach constructs useful models that act as predictors of the performance and the effects of architectural trade-offs for component-based systems such as CORBA, COM+ and J2EE. This paper focuses on describing the issues involved in designing and executing a test suite that is efficient to characterize the behavior and performance profile of a J2EE application server product. The aims of the test suite are described, along with its design and some illustrative empirical results to show its effectiveness.

*Keywords:* Component-based system, COTS, middleware, performance modelling, prototype, empirical results

## 1 Introduction

Complexity is a fact of life in the software industry. Large software systems are inherently complex and expensive to build. Despite advances in mainstream software development technology, such as object-orientation and commercial off-the-shelf (COTS) middleware and component technologies, there seems little likelihood of the situation changing in the foreseeable future.

One of the key challenges in building large distributed systems is performance prediction (O'Neill, Leaney and Martyn, 2000). Performance and scalability are major

issues, especially with complex Internet applications that are susceptible to large volumes of transactions and users at peak loads. These systems need to provide fast average response times to user, and scale to handle bursts of traffic in a predictable manner. In practice, architects are currently forced to build prototypes to give some assurance that the software and hardware infrastructures that the application uses can provide the required levels of performance. Well-designed prototypes make this possible, but the situation is far from ideal.

A number of performance analysis and prediction models of server-side application exist in the literature. Menascé, Almeida (1998) and Gomaa, Menascé (2000) propose an analysis of Web applications based on workload characteristics that impact performance. They develop performance-annotated UML design models to specify the performance relevant characteristics. Reeser and Hariharan (2000) use an end-to-end analytic queuing model of web servers in distributed environments. Petriu, Amer, Majumdar and Abdull-fatah (2000) apply the Layered Queuing Network model to predict middleware performance. However while these models are useful to specify architectural characteristics and capture key performance parameters, most of the models are purely simulated. Simulation tends to be rather inaccurate because it ignores the essential properties and details of specific COTS middleware and components. In fact these low-level details greatly impact the server-side application performance. A simple test case can demonstrate the performance difference is significant for various COTS technologies (Gorton 2000).

In the Foresight project, a practical solution to this problem, based on empirical testing and mathematical performance modeling, is being investigated. The approach aims to minimize the risks of an application failing to achieve the required performance levels, and reduce the effort required to experimentally discover configurations that achieve high performance. The approach constructs useful models that act as predictors of the performance and the effects of architectural trade-offs for component-based systems such as CORBA, COM+ and J2EE.

This paper focuses on describing the issues involved in designing and executing a test suite that is efficient to characterize the behavior and performance profile of a

J2EE application server product. The paper describes the aims of the test suite, along with its design and some illustrative empirical results to show its effectiveness. The paper concludes by describing the current status of the project and the many outstanding problems that remain to be solved.

## 2 Performance of Component-Based Systems

There are a number of component architectures that are widely used as the key software infrastructures for complex distributed applications. The predominant ones in use today are CORBA from the Object Management Group, COM+ from Microsoft and J2EE from Sun Microsystems. The discussion that follows is aimed at these and equivalent technologies.

Very basically, component architectures provide run-time environments that provide application level components with the many services they require to operate in a distributed system. These services typically include object location, security, transaction management, integration services, database connection pooling, and so on. The overall aim is to free application level components of the need to manipulate these services directly in their code, hence making them simpler to build and maintain. The component run-time environments, typically called *containers*, provide these services to the application components they host. Individual components are able to request specific levels of service from a container (e.g. no security, encryption, etc) declaratively, using some form of configuration information that the container reads.

There is an important distinction to be made about the type of components that make up applications built using middleware. As we've already pointed out, COTS middleware components form the infrastructure, or *plumbing* of distributed applications. In effect this infrastructure provides a distributed environment for deploying *application level components* that carry out business-specific processing.

This distinction between infrastructure level components and application level components is crucial. Application level components rely on the COTS middleware-supplied infrastructure components to manage their lifecycle and execution, and to provide them off-the-shelf services such as transactions and security. Hence, an application level component cannot execute outside of a suitable COTS middleware environment. The two are extremely tightly coupled.

An important implication of this is that the behavior of application components is completely dependent upon the behavior of the infrastructure components. The two cannot be divorced in any meaningful way – the entire application's behavior is the combination of the behavior of the application – the business logic - and infrastructure components. This scheme is depicted Figure 1.

This tight coupling of application and infrastructure invalidates traditional approaches to application performance measurement, and makes prediction of the effects of various architectural trade-offs more complex.

For example, it is no longer possible to execute the application components independently and measure their performance. Nor is it possible to manually inspect the code in an attempt to analyse performance, as the infrastructure source code is unlikely to be available.<sup>1</sup>

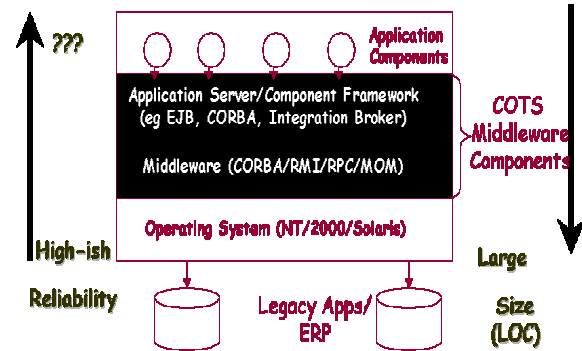


Figure 1 Anatomy of a COTS Middleware Application

This last point is absolutely crucial. CSIRO's MTE project (Gorton 2000) clearly demonstrates the performance differences between different EJB products. Figure 2 illustrates the results of executing identical application components on six different COTS components infrastructures based on the J2EE open standard. All the tests are executed on the same physical hardware and software environment, and the products are configured to achieve optimal performance on the test hardware available.

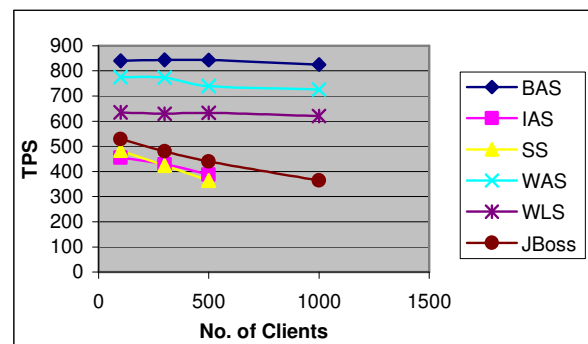


Figure 2 Performance Comparisons of J2EE Component-based Systems

The graph shows the application throughput achieved in terms of transactions per second (tps) for client loads varying between 100 and 1000. The performance differences are significant, both in the peak throughput achieved for each COTS technology, and their ability to scale to handle increasing client loads. These differences become more even more significant as the same test case is scaled out to run on more application server machines in an attempt to improve application throughput.

It therefore should be quite apparent that any performance prediction approach that does not take in to account differences between actual products is doomed to failure.

<sup>1</sup> Even if it were, the complexity of the infrastructure code would make this infeasible.

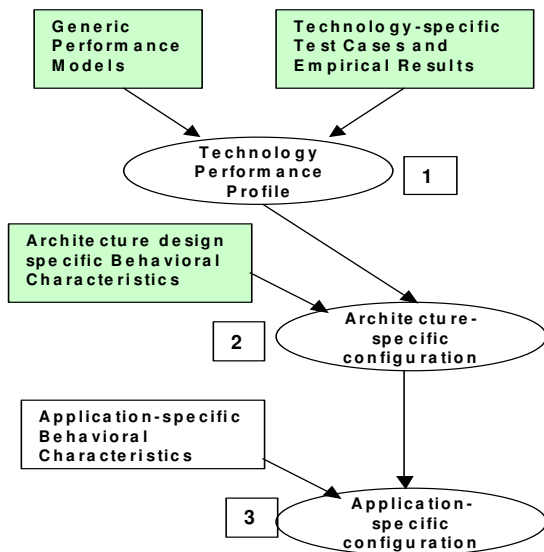
Hence new, more sophisticated approaches are required that cater for the behavior of real technologies.

Given all these issues, software engineers in practice must resort to experimentally discovering application configurations that provide acceptable levels of performance. This can be a time-consuming and expensive process, as it requires detailed measurements to be recorded from test runs across a number of different configurations. As COTS middleware technologies are highly configurable, often with tens of inter-related configuration options, this is rarely a trivial performance tuning exercise.

### 3 The ForeSight Approach

As depicted in Figure 3, the performance prediction methodology has three aims.

The first is to create a COTS product-specific performance profile that describes how the various components of the middleware product affect performance. This profile is aimed at analysing the behavior and performance of a middleware product in a generic manner that is not related to any particular application requirements. Using this profile, it should be possible to use a set of generic mathematical models to predict the behavior of the middleware infrastructure under various configurations.



**Figure 3 Performance Prediction Methodology Outline**

The second aim is to construct a reasoning framework for understanding architectural trade-offs and their relationships to specific technology features. This reasoning framework provides the architect with insights into how the different quality attributes of the application interact with each other. It aims to help the architect reason about the effects of their architectural decisions and the effects of these on application performance and scalability.

The third and final aim is to create an application-specific configuration that takes in to account the behavioral characteristics of the application at hand. The application architect describes the application behavior in terms of client loads, business logic complexity, transaction mix, database requirements, and so on. By inputting these parameters in to the generic performance models, it is possible to predict the application configuration settings required to achieve high performance.

The remainder of this paper focuses on the first step of this methodology, and specifically the design of a set of test cases that can be used to characterize the performance of a COTS middleware technology. In particular, we use a J2EE/EJB application server technology as an example.

### 4 Test Suite Objectives

There are three major aims that any test suite must satisfy in order to be useful for performance prediction:

- 1) The test suite must cover the major core and optional components of a middleware technology that are typically used in applications.
- 2) The results must show the effects of introducing various components in to an application. For example, if an application is made transactional, what is the effect of this on performance compared to a non-transaction application?
- 3) The test suite must be executable and analyzable in an economically viable manner. It cannot take months to complete, otherwise the technology will have moved on to a new version.

There are two other points to note. First, due to the very nature of COTS middleware technology, any test suite will be technology specific. For example, a test suite for J2EE technologies will be different to a test suite for COM+ technology. However, it should be possible to use the same test suite for all products that adhere to the J2EE specification. In addition, the same fundamental design for the test suite should apply to different technologies. It will be the precise implementation details that differ.

Second, the test suite should reveal how the COTS middleware under test performs in the most basic, generic sense. If we can understand and model the behavior solely of the middleware infrastructure itself, it should then be possible to augment this with application-specific behavior (Grundy, Cai and Liu 2000). This implies that the test suite should have the minimum application layer possible to exercise the middleware infrastructure in a meaningful and useful way.

In the discussion that follows, we focus on a J2EE/EJB application server technology as a concrete example.

### 5 J2EE Test Suite Design

Central to the J2EE specification is the Enterprise JavaBeans (EJB) framework. EJBs are server-side components, written in Java, that typically execute the application business logic in an N-tier application. An EJB container is required to execute EJB components.

The container provides EJBs with a set of ready to use services including security, transactions and object persistence. Importantly, EJBs call on these services declaratively by specifying the level of service they require in an associated XML file known as a deployment descriptor. This means that EJBs do not need to contain explicit code to handle infrastructure issues such as transactions and security.

An EJB container also provides internal mechanisms for managing the concurrent execution of multiple EJBs in an efficient manner. EJBs themselves are not allowed to explicitly manage concurrency, and hence must rely on the container for efficient threading and resource usage, including memory and thread usage for application components (EJBs) and database connections.

Figure 4 illustrates the major components in a typical J2EE application server product.

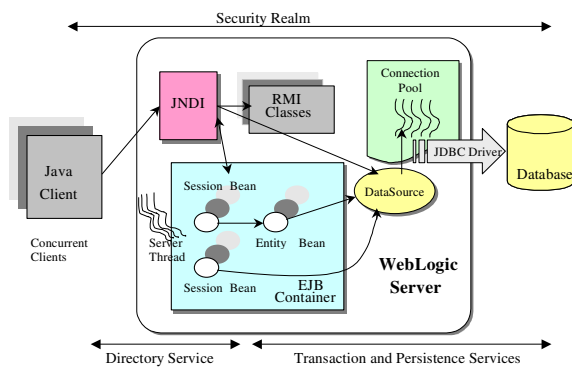


Figure 4 WebLogic Server Application Architecture

## 5.1 Basic Test Case

In order to examine the infrastructure component quantitatively, and independent from the application behaviour, we use a simple *identity* application as basic test structure. The *identity* application has several important characteristics that make it an ideal test application for examining and exposing infrastructure quality. The *identity* application has the following characteristics:

- 1) There is only one table in the relational database containing the representative business data.
- 2) The table contains 2 fields only: a unique identifier and a value field.
- 3) A single application component (e.g. an EJB) with *read* and *write* methods in its interface.
- 4) The *read()* method simply reads the value field from the database, given the identifier.
- 5) The *write()* method increments the value field in the database associated with the identifier.

Each client application has a unique id, which it uses in read and write requests. This is the client's *identity*.

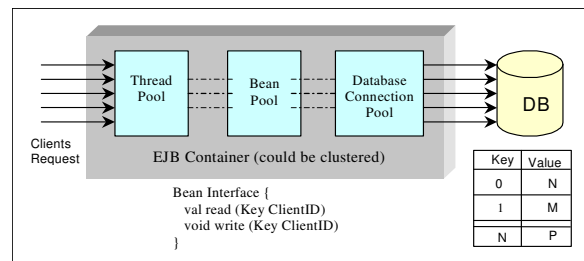


Figure 5 J2EE/EJB Test Suite Design

As a comparison, Sun Microsystems has released the Ecpert benchmark for testing EJB container performance. It simulates manufacture/supply chain operations and has complex business logic. In such a system, the performance depends both on the J2EE application server infrastructure and the application itself. Therefore, it is hard to attribute the observed performance properties to either the application server infrastructure or to the application components.

Using the *identity* application, we can thus remove any unpredictability in timing due to business logic. All client threads are randomly allocated a unique identity from 1 to the number of rows in database at run-time. Each then repetitively calls *read()* and *write()*, supplying its identity to each call. The bean's business logic uses the supplied identity key to access a single row in the database. A read request simply returns the value stored in the client's row. A write request adds one to the value stored in the client's row. The benefit is that this isolates the effect of application behavior on system performance. Hence it helps us to focus on the "pure" behavior of EJB container, namely the infrastructure code and its effect on performance.

We have also written a performance monitor program that displays in real time the number of transactions per second that the system is executing. This allows us to monitor the system's behavior as the test progresses, and clearly indicates if problems occur during a test run. Collectively, these measures make it possible to gain an excellent understanding of how a technology behaves.

## 5.2 Adding COTS Component Complexities And Setting up Test Case Parameters

Our aim is to test the basic skeletons of the infrastructure COTS component firstly using the *identity* application. Then, additional service components such as transaction and/or persistence services can be gradually added onto the basic test case. Table 1 lists the combination of architecture design and service component configuration for a J2EE application server. They cover the basic concerns of non-functional requirements when building enterprise systems, such as the modifiability, security, reliability, usability and so on (Klein, Kazman, Carriere, Barbacci Lipson 1999). Therefore it is possible to derive optimal application configuration and performance model based on these pre-defined scenarios.

Architectural alternatives	Service component configuration
Stateless session bean only	Basic JNDI directory service Bean managed transaction with JTA code No persistence; No security
	Basic JNDI directory service Container managed transaction No persistence; No security
Stateless session bean as facade for entity bean	Basic JNDI directory service Container managed transaction Container managed persistence No security service
	Basic JNDI directory service Container managed transaction Bean managed transaction Security service

**Table 1: Service Component Configuration**

This multi-staged testing approach enables us to observe the differences between a basic system (with no additional services) and another that utilizes some combination of additional service components. Between each test, we maintain all environmental variable as constant, while injecting certain variations in external stimulus test values, as shown in Table 2.

Parameter type	Sample test parameters
External variable stimuli	Client request load, client request arrival rate, transaction types, transaction request frequency
Configurable system parameters	Thread pool size, database connection pool size, application component cache size, JVM heap size
Measurable /observable parameters	Throughput (transactions per sec), average client response time, total client test time, run-time throughput
Deducible system properties	Optimal thread pool size for achieving maximum throughput, optimal database connection pool size for achieving maximum throughput, optimal application component cache size for achieving minimal response time

**Table 2: Parameter Types and Sample Test Cases**

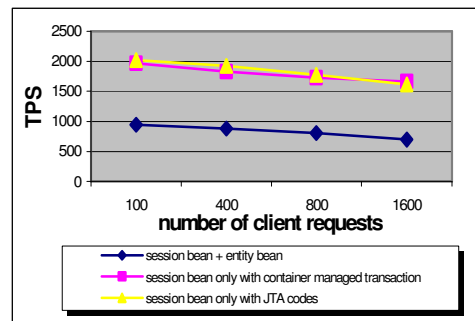
In the results that follow, the same physical test platform was used to run all test cases. The platform involved three quad-processor Windows 2000 machines, one for the client, one for the application server, and one for the database. A 100 Mbit network connected the machines.

### 5.3 Effects of Service Component Configuration

In J2EE applications, two main application architectures are typically used in server components. First, a session EJB is invoked by a client, and the session bean accesses the database directly using JDBC calls. Alternatively, an EJB entity bean can be introduced to separate the business data from the business logic in the session bean. The entity bean represents a clean object-oriented programming abstraction. Entity beans are used to

encapsulate business data in memory such that a session component (e.g. stateless session bean) can then perform the business logic on behalf of the client by accessing and manipulating business data stored in entity beans. Also, the principle of separation of concerns is observed as the business data is encapsulated in entity beans, and that the implementation of session beans is not littered with database access code.

However, the benefits of the highly modifiable entity bean architecture come at the cost of performance. A performance penalty is introduced by the creation and management costs for the entity bean. In addition, the EJB container has to synchronize the data in the entity bean with the underlying data store, which can incur expensive I/O operations.



**Figure 6 Entity Beans versus Session Beans**

Figure 6 shows the performance in term of transaction per second (TPS) for two these two architectures. The results are obtained from running the *identity* test case. The performance differences are clearly evident. In fact, the performance of the entity bean architecture is less than 50% of the performance of the session bean only architecture.

Figure 6 also compares the performance of two different transaction management strategies that are available for the session bean only architecture. Container Managed Transactions (CMT) uses the Java Transaction Service (JTS) to control transaction processing and the demarcation of transaction boundaries declaratively in the deployment descriptor files. The EJB container then automatically manages transaction outcomes. The alternative is Bean Managed Transactions (BMT), in which the transaction boundary is controlled programmatically by calling the Java Transaction API (JTA), as in the following:

- 1) UserTransaction.begin()
- 2) UserTransaction.commit()
- 3) UserTransaction.rollback()

Since JTA is high-level application programming interface for underlying JTS routines, we expected that the difference in performance would be relatively small. As depicted in Figure 6, this hypothesis is confirmed, with the performance difference between BMT with JTA codes and CMT about 3%.

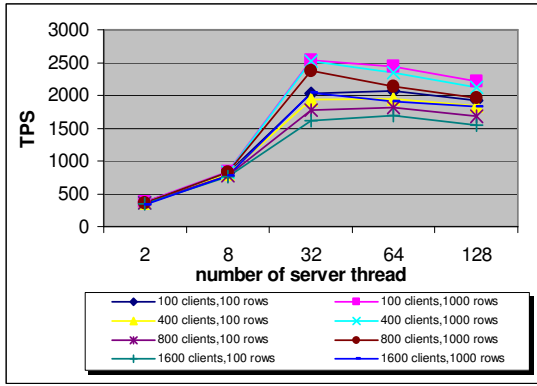


Figure 7 Session bean only, bean managed transactions

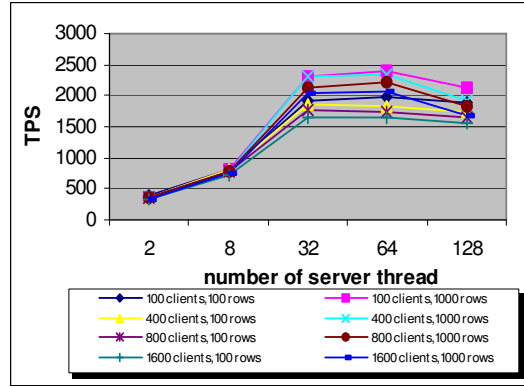


Figure 8 Session bean only, container managed transactions

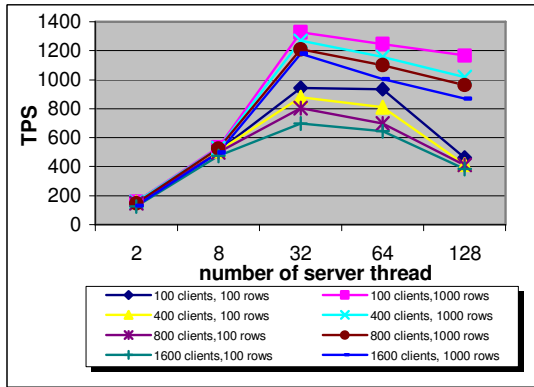


Figure 9 Entity bean and session bean

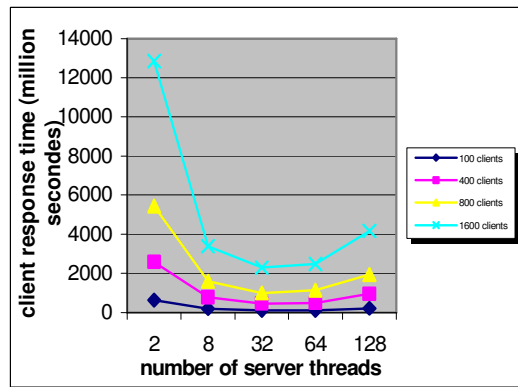


Figure 10 Response Time: Entity bean and session bean

#### 5.4 The Effect of Server Thread

EJB containers typically allow the explicit configuration of the number of threads that the container uses to execute application EJB code. This configuration therefore represents an important tuning option. Too few threads will limit performance by serializing much of the application processing. Too many threads will consume resources and increase contention, again reducing application performance. Finding the balance is typically done experimentally.

Figure 7 shows the performance of the session bean and BMT test case as the number of threads in the container is varied from 2 to 128. Figure 8 depicts similar data except container managed transaction are used, and Figure 9 represents the performance observed using both session and entity beans. In all tests, the size of the database connection pool is set to the same value as the size of the thread pool. This is a vendor recommended setting.

Despite the architectural variation, the results produce the same trend. The performance increases from 2 threads to 32 threads, stabilizes around 32 to 64 threads, and gradually decreases as more threads are added. Hence it appears that for the application architectures represented in this test case, the thread count is an independent variable. Of course, more data points, such as 24 and 48 threads, are needed before the optimal number of threads can be predicted. The results are however still revealing.

#### 5.5 The Effect of Database Contention

The *identity* test case is deliberately designed to avoid database contention. This occurs when two concurrent transactions attempt to access the same data items in the database. Database contention heavily depends on the business logic of an application and the database design, as well as configuration parameters such as the number of application server threads. To explore the effects of database contention, the *identity* test case can be configured to:

- 1) Limit the number of rows available in the database table to a specific maximum value
- 2) Force each client request to randomly choose a database row to access in a transaction within the range available

In such a scenario, let  $R$  be the number of rows in the database table. The possibility of database contention occurring is  $\kappa/R$ . When the transaction type is defined by the business logic and the number of server thread is fixed,  $\kappa$  is constant. Hence the possibility of contention occurring is affected by  $R$ . To validate this, the tests varied the value of  $R$  from 100 to 1000 and left all the other parameters unchanged. When the number of rows in database increases, the possibility of database contention decreases, and thus the overall performance improves. The effect of database contention leads to performance that is between 20% and 49% lower than that shown in Figures 7, 8 and 9.

## 5.6 The Effect of Client Request Load

Client request load is a key characteristic for performance prediction. As more concurrent client request arrive at an application server, more contention will be incurred for server threads to process these requests. Consequently there will be longer request queue length in the application server, as requests wait for service. This all leads to an increase in client response time. This is clearly shown in Figure 10.

Figure 11 depicts application performance for the session and entity bean architecture, with 32 server threads and a limited 1000 database rows in the test configuration. The overall system throughput shows a peak of 1327 transactions per second with 100 clients. This declines slowly to 1210 tps with 800 clients and 1179 with 1600. The additional load being placed on the WLS container by the heavier client request load causes this decline. Importantly, the decrease in performance is very small from 100 to 800 clients (approximately 8%) and linear. This is a good result, and indicates that the internal EJB container architecture will scale well to handle greatly increasing client loads.

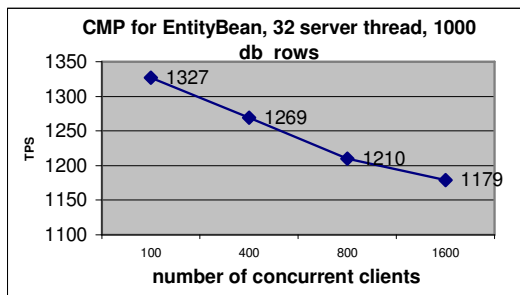


Figure 11 TPS with varied client request load

## 6 Future Work

This paper describes results that form part of the first step of the ForeSight approach described in section 3. The results presented are a small subset of the tests needed to fully characterize the behavior of an application server product. Further tests are needed to extend the test suite to cover more technology properties such as scalability, clustering, distributed transactions, state management and so on. These results can then be used to calibrate generic performance models so that predictions can be made specifically on the performance of the WebLogic Server product.

Next, the full test suite will be run on another J2EE application server product, and on Microsoft's COM+. The results from these tests will enable us to perform some extensive experiments in performance prediction, and test the boundaries and usefulness of the models.

## 7 Conclusion

This paper has described the inherent difficulties of predicting the performance of N-tier enterprise applications built using COTS middleware components. The key problems revolve around the diversity and complexity of the COTS middleware products that are used to build large-scale distributed applications. In practice, these problems force software architects to experimentally discover

application configurations that provide the desired levels of performance. This is a time-consuming, expensive and non-trivial exercise, and must be done late in the project life-cycle when the application has been built.

The ForeSight approach explained in this paper explicitly recognizes these problems, and provides a potential solution based on empirical testing and mathematical modeling. The models describe generic behaviors of application server components running on COTS middleware technologies. The parameter values in the models are necessarily different for each different product, as all products have unique performance and behavioral characteristics. These values must therefore be discovered through empirical testing. To this end, a set of test cases are defined and executed for each different COTS middleware product. The results of these tests make it possible to solve the models for each product, so that performance prediction for a given product becomes possible.

This paper has shown that the simple test case used for gathering empirical results is effective. The analysis of the results obtained clearly differentiates between different application architectures and COTS middleware configurations. The data therefore promises to provide rich content with which to calibrate the performance models we are deriving.

In the medium term, there remain a number of complex problems to solve. Incorporating application-specific behavior in to the equation in a simple and practical manner is still an open problem. This is necessary to make this approach useable in wide-scale engineering practice. It also remains to be seen how far the results from the empirical testing can be generalized across different hardware platforms, databases and operating systems. This is important, as it profoundly affects the cost of performing the test cases and obtaining the empirical results. It may be that, if a huge number of test cases need to be executed on different platforms for every product, this approach may be economically impractical in a technological environment of rapid change and evolution.

In the longer term, this work is a step on the road to providing real-time performance monitoring and truly adaptive application behavior. In such a scenario, applications would dynamically monitor and measure aspects of their behavior, and independently feed these measures in to generic behavioral models. The results could then be used to adaptively self-configure the underlying COTS application infrastructure. This instantly solves the problems of generality of empirical results and economic feasibility, and should lead to higher performance applications that can be engineered at much lower cost.

## 8 References

O'NEILL, T., LEANEY, J., MARTYN, P. (2000): Architecture-based performance analysis of the

- COLLINS class submarine Open System Extension (COSE) Concept Demonstrator (CD). Mining association rules between sets of items in large databases. *Engineering of Computer Based Systems, 2000, Proceedings of the Seventh IEEE International Conference and Workshop on Engineering of Computer Based Systems*, 26-35.
- Gorton, I. (2000): *Middleware Technology Evaluation Series*. CSIRO, Sydney.
- MENASCÉ, D.A., ALMEIDA, V.A.F. (1998): Capacity Planning for Web Performance: *metrics, models, and methods*. Upper Saddle River, NJ: Prentice Hall.
- GOMAA, H., MENASCÉ, D.A. (2000): Design and performance modeling of component interconnection patterns for distributed software architectures. *Proceedings of the second international workshop on Software and performance*. Ontario, Canada, 207-216, ACM Press.
- REESER, P., HARIHARAN, R. (2000): Analytic model of Web servers in distributed environments. *Proceedings of the second international workshop on Software and performance*. Ontario, Canada, 158-167, ACM Press.
- PETRIU, D., AMER, H., MAJUMDAR, S. ABDULL-FATAH, I. (2000): Using analytic models predicting middleware performance, *Proceedings of the second international workshop on software and performance*. Ontario, Canada, 189-194, ACM Press.
- GRUNDY J.C., CAI, Y. and LIU, A. (2001): Generation of distributed system test-beds from high-level software architecture descriptions, *Automated Software Engineering*, San Diego, USA, 26-29, IEEE CS Press.
- KLEIN, M., KAZMAN, R., CARRIERE, B.J, BARBACCI, M., LIPSON, H. (1999): Attribute-based architecture styles, *Proceedings of the First Working IFIP Conference on Software Architecture*, San Antonio TX, USA 225-243.