

Evaluating the Scalability of Enterprise JavaBeans Technology

Yan (Jenny) Liu*, Ian Gorton**, Anna Liu*, Shiping Chen***

*School of Information Technologies, University of Sydney, Sydney, Australia

**Pacific Northwest National Laboratory, Richland, WA 99352, USA

***CSIRO Mathematical and Information Sciences, Sydney, Australia

jennyliu@it.usyd.edu.au, ian.gorton@pnl.gov, annali@microsoft.com, shiping.chen@csiro.au

Abstract

One of the major problems in building large-scale distributed systems is to anticipate the performance of the eventual solution before it has been built. This problem is especially germane to 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 individual application diversity, software architecture trade-offs, COTS component integration requirements, and differences in performance of various software and hardware infrastructures. In this paper, we describe the results of an empirical investigation into the scalability of a widely used distributed component technology, Enterprise JavaBeans (EJB). A benchmark application is developed and tested to measure the performance of a system as both the client load and component infrastructure are scaled up. A scalability metric from the literature is then applied to analyze the scalability of the EJB component infrastructure under two different architectural solutions.

1. Introduction

Scalability is a measure of the flexibility of a system to adapt readily to intensity changes of usage or demand, while still meeting the predefined business objectives [1]. Scalability is a particularly important issue in Web-enabled information systems. Industry folklore is rife with stories of Internet e-business sites breaking under unexpected client surges in demand. In this context therefore, the importance of performance and scalability cannot be understated. Achieving scalability involves considering the overall system properties, such as availability, reliability, mobility, security and performance. It remains a significant challenge for engineers constructing a large distributed system.

Sun Microsystem's Java 2 Enterprise Edition (J2EE) platform attempts to simplify distributed application programming as well as providing the foundation for scalable solutions. J2EE defines the Enterprise JavaBeans (EJB) specification, which is essentially a

component framework for server-side components that are hosted in a run-time environment called an EJB container. Clustering EJB containers facilitates scale-out of an EJB solution. Clustering is consequently a key service in ensuring high performance and scalability of a J2EE application [3].

Over 30 product vendors provide implementations of the J2EE/EJB specification. As each of these products is designed and implemented differently, they show radically different performance profiles [2]. To complicate this technology space further, product vendors of J2EE products typically provide different infrastructure options and tuning techniques to address the requirements of performance and scalability. For example, most J2EE application servers provide a product-specific mechanism for clustering J2EE servers.

In this paper we propose an approach to measure the performance and scalability of COTS EJB technology and illustrate it using a leading J2EE application server¹ product. In section 2, we introduce the architecture of EJB clustering. In section 3, a benchmark application is described and in section 4 the experimental methodology is addressed. Next, a scalability metric from the literature is applied to the benchmark results obtained from the J2EE product. Section 5 represents the scalability metric and maps EJB clustering scenarios to the scaling strategies covered by the metric. The resulting scalability measure is analyzed in section 6, and related work and conclusions presented in sections 7 and 8.

2. EJB Clustering

J2EE products support EJB clustering services to deliver scalability and high availability. A cluster is a group of EJB servers and machines working together to transparently provide application services. Approaches to clustering vary considerably across vendor implementations and hence are a major point of product differentiation. In this paper, the latest version of one of the leading commercial J2EE application server products is used for testing and consequent scalability analysis.

¹ Licensing restrictions mandate that the product cannot be named.

The product tested in this paper implements clustering in this manner. It is using replica-aware stubs and a shared global directory (JNDI) tree. If an EJB object is clustered, its instances, called *replicas* are deployed on all the servers participating in a cluster. The replica-aware stub represents a collection of replicas distributed on the server instances. When an individual server starts up, it binds the replica-aware home stubs to its local JNDI tree. Then it puts its local JNDI tree into the shared global JNDI tree. Each server communicates with others in the cluster to keep their own JNDI tree consistent. This means EJB components can be replicated across the individual EJB servers. Clients download replica-aware stubs from the JNDI and use them to lookup the object on the server. The replica-aware stub can then be used by a load-balancing algorithm to dynamically choose a server on a per-call basis. This is done transparently to the client application. Figure 1 illustrates this clustering architecture.

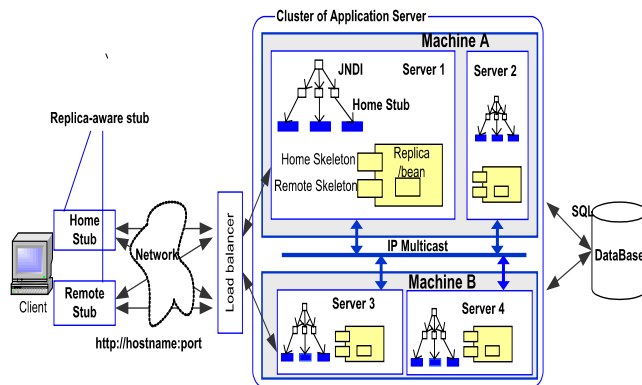


Figure 1 EJB cluster architecture

3. Benchmark Application Design

3.1. Identity application characteristics

A benchmark application produces measurements that are particular to that benchmark. Consequently, the characteristics of a benchmark application will greatly influence the scaling strategy of the system under test. [5] and [6] cogently discuss the inadequacy of industrial standard benchmarking for Java applications. The authors believe that no industry standard benchmark can accurately represent the behavior of all different types of applications, and thus the results tend to be misleading. The major J2EE benchmark, ECperf™ [7] includes relatively complex business logic in the application components, and is heavily reliant upon the performance of the external database system used in the tests. This makes it difficult to distinguish application server specific performance issues from those occurring in the database. As we are interested in assessing the scalability solely of the EJB infrastructure provided by an

application server, we deliberately avoid using ECperf at this stage. The benchmark application used in this paper is therefore simplified to a set of core application server elements.

The benchmark we have devised has been named the *identity* application. It is basically a ‘ping’ operation that exercises the whole EJB infrastructure. The identity application has several important characteristics that make it an appropriate test application for examining and exposing the implementation quality and behavioral characteristics of a middleware infrastructure. These are:

- The associated database has a table containing 2 fields only: a unique key identifier and a value field.
- A single externally visible application component (e.g. an EJB) that has *read* and *write* methods in its interface. The *read ()* method simply reads the value field from the database, given the identifier. The *write ()* method increments the value field in the database associated with the identifier.
- Each client application is randomly allocated an identifier, which it uses in read and write requests. This is the client’s run-time identity.

The consequences for the empirical results we obtain from the identity application are as follows:

- Simplified business logic means the benchmark application is essentially free of unpredictable timing due to external interactions. This makes it easier to attribute any observed behavior to either the application server implementation or to the application components.
- The empirical results are technology and platform specific, but with absolutely minimal dependence on the application components. This means that the performance measures are essentially capturing the performance of the component infrastructure itself. The effect is to *calibrate* the infrastructure in the test environment. Adding further complexity in terms of application components or J2EE services can only result in *lower* performance.
- The results that we obtain with the *identity* benchmark are the best possible for an application with the same basic characteristics of querying and updating a database, running in the same test environment.

The *identity* application also provides the basis for incrementally incorporating features such as more complex business logic and additional services such as distributed transactions and component persistence [9]. Incorporating application-specific characteristics into the benchmarking profile in a meaningful way remains a goal of our future work.

3.2. Client simulation

The test case in this paper is aimed only at measuring EJB server performance and scalability. This comprises

one of the layers in a typical multi-tier architecture for an enterprise application. In such architectures, each tier basically plays the role of a *multiplexer*.

For example, there may be 100,000 people connected to a web server, most thinking, and some submitting requests and waiting for responses. Perhaps only 1000 of these requests can be processed simultaneously by the servlet engine in the Web server, with all other outstanding requests queued. The 1000 active servlets in the web server are therefore the direct, or *proxy* clients of the next tier, namely the EJB application server tier [15] that is the focus of this work. As the EJB tier is responsible for business logic execution, it is generally the most complex computationally. Consequently it becomes the limiting factor in the performance of the overall multi-tier application.

The ‘think time’ of the EJB *proxy* clients is zero when the Web server engine is at full capacity. They receive and process requests as quickly as they can, and immediately process another that is waiting for service. This assumption will always hold when the Web server tier is fully utilized.

The clients in our tests model this behavior. We use a standalone Java application to represent the EJB proxy clients. The number of concurrent client requests is scaled to simulate the load on the EJB server. Once a transaction is complete and the results are returned from the EJB server components, each simulated proxy client immediately starts the next selected transaction.

The simulated client proxy response time R_m is measured by the real, or wall-clock time spent processing the request, rounded to millisecond precision. We also estimate the client side response time R_c with the formula,

$$R_c = N / \lambda(N)$$

where N is the number of concurrent clients and $\lambda(N)$ is the throughput of N clients in term of transaction per second.

R_c overestimates the client side response time because it includes the processing time at a client to generate the next request. Also the average queuing length in the server side is smaller than N , which means the server delay is smaller. However, under all client request loads from 100 to 5000, the difference between R_c and R_m is within 10 milliseconds, which is essentially not significant given the inherent error in Java timer resolutions.

3.3. Architecture alternatives

The identity application can be implemented using alternative EJB architectures. In this paper, we use two common alternatives, namely a stateless session bean

architecture, and a stateless session bean as a facade for entity bean architecture².

With the stateless session bean architecture, the session bean includes both the business logic and database access, as shown in Figure 2. Container managed transactions (CMT) are used in these tests, so that the session bean code does not need to explicitly demarcate the transaction boundary [8].

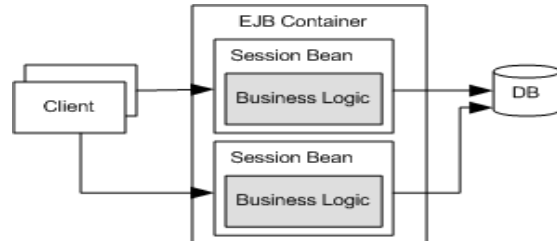


Figure 2 Session bean architecture

With the session bean + entity bean architecture, the session bean still implements the business logic but does not contain database access code, as shown in Figure 3. The session bean uses an EJB entity bean as a Java object representation of underlying relational database. Container managed persistence (CMP) automatically ensures that the data encapsulated in entity beans is up to date and correct with respect to the persistent data in the database. The transaction context propagates from the session bean to the entity bean, again using CMT.

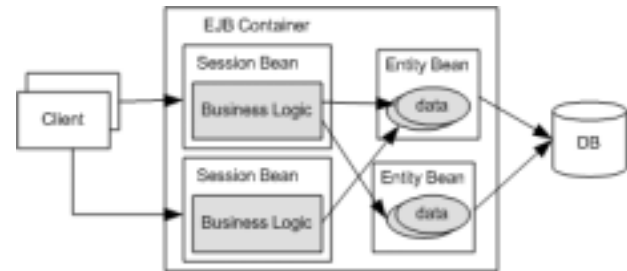


Figure 3 Session bean +entity bean architecture

4. Experiment methodology

4.1. Test setup

The test environment comprises four machines connected by a 100Mb isolated network. One 4-CPU machine is used for the clients, one 8-CPU machine for the database server and the other two 4-CPU machines for the EJB servers. The application server and Oracle 8.1.7 database run on Windows 2000 server. The JVM is Sun’s JDK 1.3.1.

² We use the notation session bean + entity bean to represent the architecture of session bean as a façade of entity bean in the rest of this paper.

Extensive tuning is carried out to ensure near optimal performance is achieved. For example, the application server under test is started with the hotspot option for the JVM. The JVM heap size is tuned to minimize the garbage collection effect during the test measurements.

4.2. Measurements

The empirical testing uses client loads increasing from 1 to 5000. The low client loads were used to decide the base load for the scalability calculation. As shown in Figure 4, the measured throughput for both test architectures increases up to client loads of a 100. At this level, the CPU utilization on the servers begins to approach saturation. It was therefore decided to use the workload generated by 100 clients as the base level for scale calculations. Below this level, the server machines are essentially under utilized.

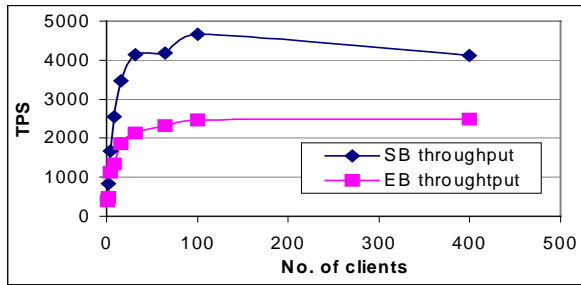


Figure 4 Throughput of 1 to 400 clients

5. Scalability metric

A framework for evaluating scalability is proposed in [10]. The scalability framework comprises four aspects, a *scaling strategy*, a *scale factor*, a *scale variable* and a *scale enabler*. The scale factor represents one dimension of the system scale, such as the number of users, CPU load, process replication or thread pool size. The scaling strategy is controlled by the scale factor. When the scale factor changes, the scaling strategy determines the scaling variables, which are required to achieve optimal configuration by tuning the value of scale enablers. The scalability is measured by *productivity*, $F(k)$, where k is the scale factor. The scalability metric at two different levels of the scale factor is the ratio of their productivity $F(k)$.

$$F(k) = \lambda(k) * f(k) / c(k)$$

$$S(k1, k2) = F(k2)/F(k1)$$

where:

$\lambda(k)$ is the system throughput.

$c(k)$ is the cost at scale factor k , it is the sum of purchase cost and maintenance cost.

$f(k) = 1/(1 + T'/T)$, where T' is the average response time and T is the target value. $f(k)$ is the value function that is introduced to measure the quality of service. It

may be any of the system measurements, such as response time, throughput and so on.

Productivity $F(k)$ takes advantages of the flexibility of value function $f(k)$. Thus it is possible to use the productivity to cover the impact of different quality attributes on performance. By means of the value function and the cost function, the metric actually evaluates the system scalability in a cost-effective way.

The main concern for us to adopt this scalability metric is that it is within the context of scaling strategies, which are application and technology specific. It combines the capacity, QoS and cost together. Therefore it is possible to map the tuning options of EJB applications to the scalability space in a sensible way.

In scaling out an EJB application, the scale factor k represents the number of machines introduced into the system under testing. Now, $\lambda(k)$ is measured for each scale factor k during testing. T' is the average response time at the client. It is measured from the client side wall-clock time. The target time T is determined as 2 seconds.

As increasing numbers of machines are introduced into the cluster, the cost of hardware, software licenses, and maintenance increases. Based on the test environment used in this paper, the cost of introducing each machine into a cluster is US\$35,000.

$$c(k) = 35,000 * k$$

Now, $\lambda(k)/c(k)$ is the transactions per second per dollar in our case.

The experiments in this paper are primarily interested in evaluating scale-out across 2 clustered server machines. This is achieved by planning the scale strategy under three scenarios as follows:

A single server instance on a single machine

Concurrent clients send requests to application components running on the application server deployed on a single machine. A number m threads are specified in the application server thread pool and these are pre-created when the server starts, Figure 5. During the test, a single thread handles each client request. If the number of request exceeds m , the additional requests wait in a request queue managed by the application server. The queue length consequently increases as more simultaneous requests arrive. The thread pool size in a single server is therefore an important tuning option and it can be used as a scale enabler.

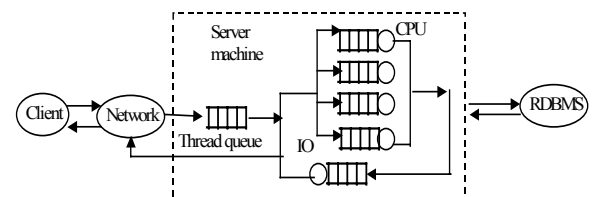


Figure 5 One server instance on one multi-processor machine

Multiple server instances on a single machine

It is sensible to start multiple server instances in a cluster on a multi-processor machine. Each server instance has its own dedicated thread queue. The number of server instances can be the scale enabler. Processors are allocated to each server application by the operating system. All the server instances in the cluster share other resources, such as network and memory. The other scale enabler is thread pool sizes per server instance. This is depicted in Figure 6.

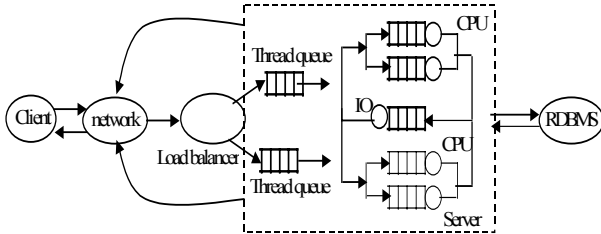


Figure 6 Clustering across a single multi-processor machine

Multiple server instances across two machines

As shown in Figure 7, in this scenario the processing capacity of the application is effectively doubled. Each machine can accommodate multiple server instances as in the second scenario. There is no difference in deployment procedures and the scale enablers are the same as the second scenario.

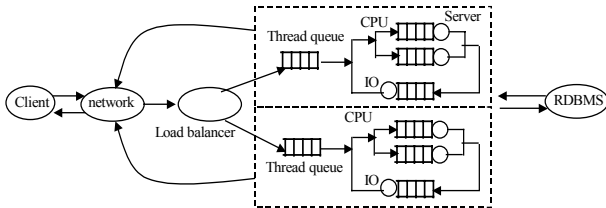


Figure 7 Clustering across multiprocessor machines

A point to note is that database replication is not selected as a scale enabler in these tests. A single database machine is used even as the application server configuration is scaled out. Obviously the database will eventually become a bottleneck as its capacity is saturated by more and more concurrent requests. However, the test environment used ensures that this is not the case by using a highly optimized Oracle 8.x instance running on a high performance 8 CPU machine. In no tests is the database capacity even near saturation, and hence the performance results reflect the pure performance of the application servers under test.

6. Results analysis

We analyze the experiment results for both session bean architecture and session bean + entity bean architecture in the same order as the scaling strategy discussed in section 5. For each scaling strategy, we modify the following scale enablers:

- Thread pool size per server instance
- Number of server instances per machine in a cluster
- Number of CPUs allocated per server instance
- Client request load, scaled from 100 to 5000

6.1. Scaling a single server instance on a single machine

EJB containers typically allow the explicit configuration of the number of threads that the container uses to execute application 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 8 and Figure 9 show the performance of the session bean and session bean+ entity bean test case as the number of threads in the container is varied from 15 (the default setting of the product) to 64. 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.

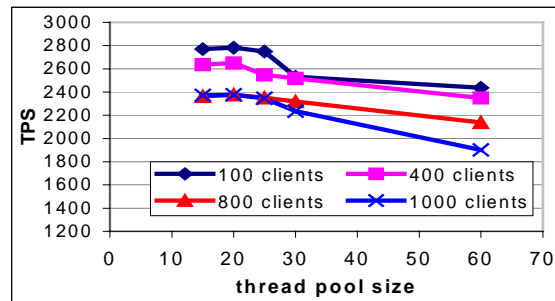


Figure 8 Session bean threads vs. TPS

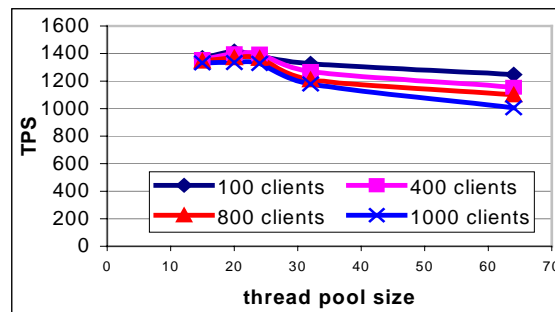


Figure 9 Session bean+entity bean threads vs. TPS

Despite the architectural variation, the results produce the same trend. The performance increases slightly from 15 threads to 20 threads, stabilizes around 20 to 24 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. It is reasonable to set 20 as the optimal thread pool size for a single server instance.

Under all client request loads from 100 to 5000, the application server CPU utilization is around 80% and database server utilization is around 30%-40%. From these observations, it is clear that the application server is a performance bottleneck [12]. However, as the application server is a black box component, it is not possible to understand its internal behavior to explain why it is not capable of saturating the CPUs available. Introducing more server instances into a cluster should improve resource utilization [12].

6.2. Clustering multiple server instances

If a single server does not fully utilize an existing machine, additional server replicas can be started. However, as more threads or server instances are introduced, they increase concurrency as well as contention. This introduces several design considerations, as follows:

- What is the optimal number of server instances in the cluster?
- Does the optimal thread pool size on a single server instance hold for all replicas in the cluster?

To investigate these issues, a series of experiments were performed, the results of which are beyond the scope of this paper. In summary, it was discovered that up to 4 server instances per machine provided throughput increases. This is in line with the vendor's recommendation of 1 server instance per CPU. The optimal number of threads for a single server instance was found to generally provide near optimal throughput when this setting was simply replicated in the clustered servers. There was however some variation across application architecture and for various client loads. When a different thread pool setting provided higher throughput, this was subsequently used in the final result gathering, and is noted in the results in the next section.

Also, when the server application is distributed across more than one machine, load balancing mechanisms must be used to ensure the additional resources on the different machines get fully and fairly utilized by the clients. Between the *proxy* client and application server components, load balancing is implemented at two levels. First, when a client 'looks up' a bean object, the request

is routed to a server instance in the cluster by the client's replica-aware home stub. Second, server component method invocations are load balanced by the client stub across the component replicas in the server instances that support that replica type. In these tests, since both the hardware capacity and server component deployment are homogenous, a round-robin load balancing algorithm was selected. This leads to even distribution of client request loads across the servers in the cluster, as demonstrated by the profiling utility output below:

```
Workload on each server:
"server-1" processed 2544 (25%) of 10115 invocations
"server-2" processed 2483 (25%) of 10115 invocations
"server-3" processed 2592 (25%) of 10115 invocations
"server-4" processed 2496 (25%) of 10115 invocations
end profiling...
```

6.3. Scalability metrics calculation

The client request load is one of the scale enablers and the scalability metric is calculated for client request loads from 100 to 5000. We only show the calculation table of 1000 clients due to the space restrictions. Note that the throughput we used for scalability metric is the optimal value shown in Figure 10 and Figure 11. Be restricted of the space, we only show the scalability metric calculation of 1000 clients for the two architectures in Table 1 and Table 2. The complete scalability results are shown in Figure 12 and Figure 13.

It can be seen that clustering multiple server instances on a single machine yields good scalability (beyond unit value) across all client request loads for both application architectures. By introducing more server instances on one machine, some internal bottleneck is removed and throughput increased with cost unchanged. Thus scalability benefits from throughput.

Adding another machine, the system capacity doubles as well as the cost. Although the throughput of clustering two machines increases up to almost 100% over that of a single server on one machine, it is the cost that drops the scalability to around unit value for 100 clients to 1000 clients. Beyond 1000 clients, the maximum scalability of using two machines exceeds unit value. This is due to the effect of the *value function*. As the throughput increases, the response time is decreased dramatically. It in turn enhances the *value function* and the scalability result.

The lower scalability of clustering two server instances across two machines than that of scaling out a single machine manifest the fact that scalability is a joint effect of system capacity, quality of service and cost in the context of sensible scaling strategies.

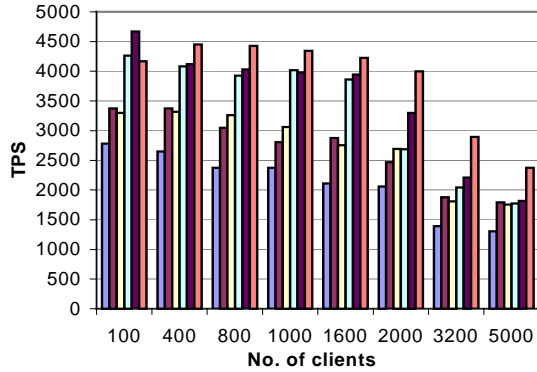


Figure 10 Session bean throughput with varying server instances

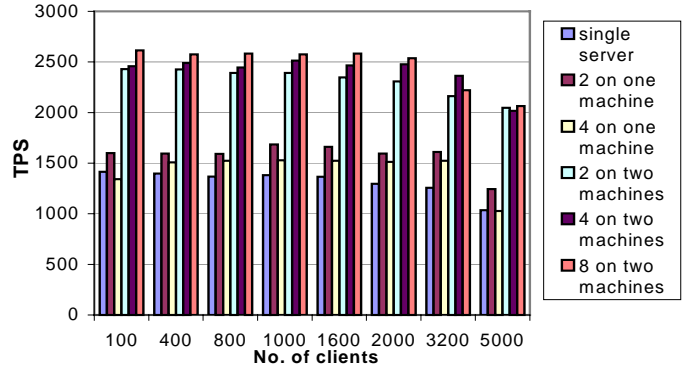


Figure 11 Session bean + entity bean throughput with varying server instances

Scale Factor k	No. of Servers	Optimal Thread Pool Size	Throughput $\lambda(k)$ in TPS	Response Time T' in seconds	Cost $c(k) = 35000 * k$ in USD	Value Function $f(k) = 1/(1+T'/T)$	Productivity $F(k) = \lambda(k) * f(k) / c(k)$	Scalability $F(k2)/F(k1)$
1	1	20	2376.4165	0.4208	35000	0.8262	0.0561	1
1	2	20	2805.4417	0.3565	35000	0.8487	0.068	1.2128
1	4	10	3058.4	0.327	35000	0.8595	0.0751	1.3389
2	2	20	4017.1	0.2489	70000	0.8893	0.051	0.9098
2	4	20	3976.067	0.2515	70000	0.8883	0.0505	0.8995
2	8	10	4344.2886	0.2302	70000	0.8968	0.0557	0.9922

Table 1 Session bean scalability metric of 1000 clients

Scale Factor k	No. of Servers	Optimal Thread Pool Size	Throughput $\lambda(k)$ in TPS	Response Time T' in seconds	Cost $c(k) = 35000 * k$ in USD	Value Function $f(k) = 1/(1+T'/T)$	Productivity $F(k) = \lambda(k) * f(k) / c(k)$	Scalability $F(k2)/F(k1)$
1	1	20	1380.7333	0.7243	35000	0.7341	0.029	1
1	2	20	1685.5834	0.5933	35000	0.7712	0.0371	1.2825
1	4	20	1528.9	0.6541	35000	0.7536	0.0329	1.1366
2	2	20	2393.3499	0.4178	70000	0.8272	0.0283	0.9765
2	4	20	2513.6501	0.3978	70000	0.8341	0.03	1.0342
2	8	20	2575.889	0.3882	70000	0.8374	0.0308	1.064

Table 2 Entity bean + session bean scalability metric of 1000 clients

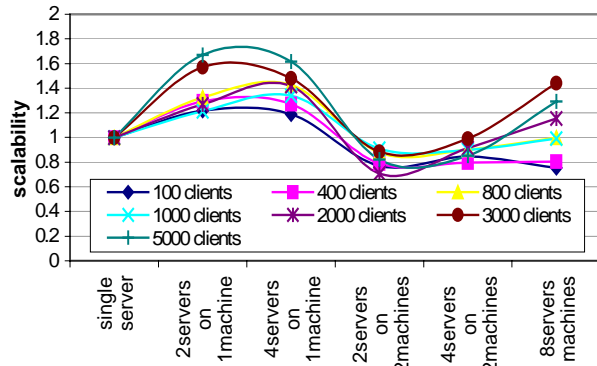


Figure 12 Stateless session bean scalability

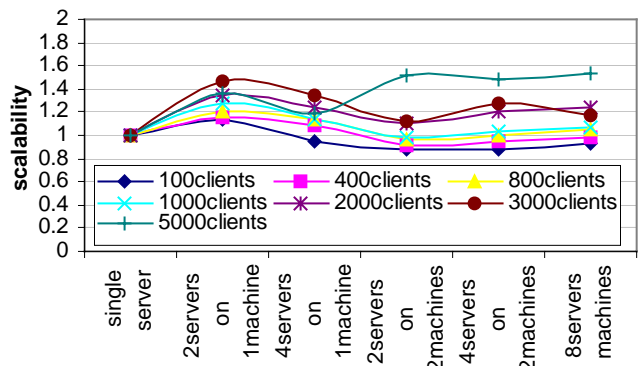


Figure 13 Session bean + entity bean scalability

6.4. Summary

The overall optimal throughput and response time of a single server, clustering on one machine and clustering on two machines are showed in Figure 14 and Figure 15. As the client request load scales from 100 to 1600 (a factor of 16), by clustering multiple server instances on a single machine the throughput is better than the throughput of 100 client requests on a single server

By introducing a new machine, we can prevent the throughput dropping when the client load scales from 100 to 5000 (a factor of 50). This can be seen for both session bean architecture and entity bean architecture. The average client response time increases linearly with the number of client requests. With two machines, only 5000 clients exceed the target time 2 seconds, 2.1 seconds for session bean and 2.4 seconds for entity bean.

This also indicates the concurrency contention can be minimized by the scaling strategies addressed above. The overhead of clustering is insignificant compared to its contribution to the overall performance and scalability.

As Figure 14 and Figure 15 clearly show, the throughput for the entity bean architecture declines much more slowly than the session bean tests, despite producing lower throughput at all client load levels. The peak throughput obtained with the session bean architecture is close to twice that obtained with the entity bean

architecture. However, with 5000 clients, the performance difference between the two architectures is less than 15%.

These results highlight the different strengths in terms of performance and scalability of these two architectural solutions. The session bean architecture is intuitively lightweight, and the very high levels of performance at low client loads, basically up to 100, clearly demonstrate this. Above this level, the performance and scalability levels diminish much more quickly than the entity bean architecture.

The entity bean architecture imposes a much greater burden on the application server infrastructure. This reduces the raw throughput levels. The peak performance however is obtained at around 1000 to 1600 clients, and in fact the throughput is very stable between 100 and 3200 clients.

Overall, the metric used represents the scalability of the system in an intuitive and useful manner. The scalability measure is useful in quantifying the effects of clustering EJB servers on the application performance. In this particular experiment, it indicates that the internal architecture of the product under test will scale well to handle greatly increasing client loads, as long as sensible scaling strategies are carefully utilized.

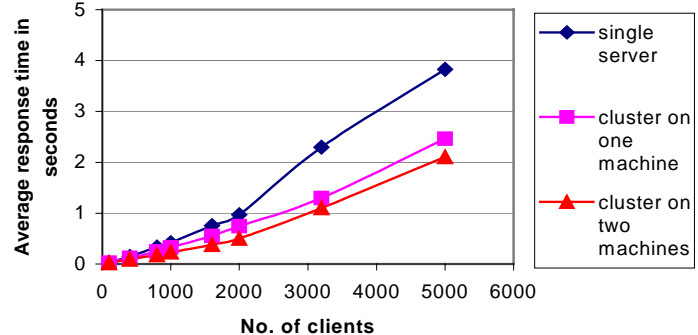
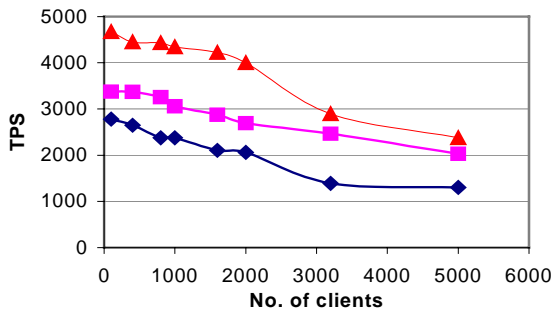


Figure 14 Session bean optimal throughput and avg. response time under three scaling strategies

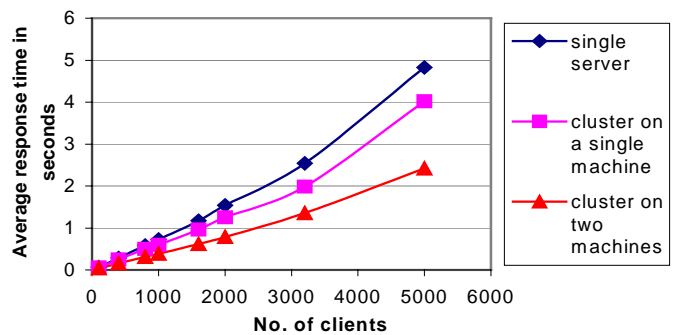
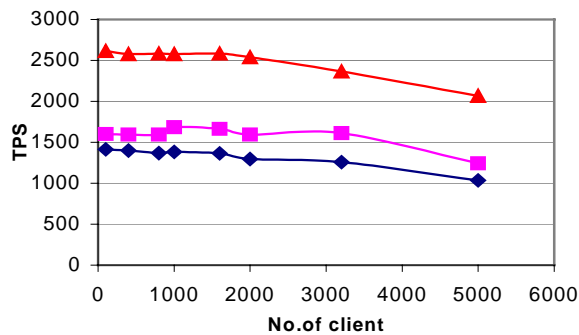


Figure 15 Session bean + entity bean optimal throughput and avg. response time under three scaling strategies

7. Related work

[1] introduces an idea to determine attributes that affect scalability and provides scaling techniques to implement scalable websites. [11] explores the workload characteristics of large commercial web sites and evaluates their effects on system performance empirically. [1] and [11] both address scalability issues in multi-tier systems supporting Web sites applications. The definition of scalability in this paper is the same as [1]. This work is concerned with the dependency of a particular EJB product behavior on the workload it must handle.

[13] demonstrates a measurement and modeling infrastructure to support system specific benchmarks for e-commerce sites. The authors have constructed a set of methods for workload characterization, modeling, and workload generation in order to stress test an application. Mean value analysis and the Layered Queuing Network (LQN) model are used for performance evaluation. The purpose of this benchmarking suite is to discover the workload mix that creates maximum utilization of a process under specific constraints, so that performance bottlenecks can be detected. Techniques for estimating the maximum utilization of a subsystem or a process in a distributed application are discussed in [13]. This approach is very close to the objectives of the work described in this paper, which focuses on how to achieve optimal performance given a particular workload mix and intensity. However it works only when the source code is available. Very detailed performance parameters are needed to feed into analytical models, which are difficult to estimate or measure from black box COTS middleware systems.

The systems research group at Rice University has established a project on *System Support for Dynamic Content Web Servers*. [16] aims to investigate the "effect of application implementation methods, container design, and efficiency of communication layers on the performance of an EJB server and the overall application". The project uses the open source JBoss and JOnAS J2EE application servers as a case study. This gives much more scope for measuring the internal behaviors of the application servers, as the source code is available for instrumentation.

[17] presents the authors' experience with running ECperf benchmark on one COTS J2EE application server. The authors discuss the performance issues related to benchmark specification and implementation and provide suggestions on how to improve performance related to database connectivity, locking patterns and asynchronous processing. Their summary about the lessons learned from the experiments is useful for tuning

application servers. However clustering is not covered in this paper.

8. Conclusions and future work

This paper has presented an empirical approach to evaluate the scalability of EJB application servers. COTS EJB technology poses a significant challenge for performance analysis and prediction of applications. This is due to their inherent complexity, and the fact that they are essentially black box components with unknown internal behaviors. Both these characteristics mean that generic models are likely to produce inaccurate and misleading results.

The effectiveness of the proposed approach and benchmark has been demonstrated experimentally using a J2EE/EJB standard compliant application server product. The results obtained from implementations of the benchmark using two common component architecture styles are analyzed using an established scalability metric. The results highlight the inherent scalability of the EJB container product up to certain limits, and how the performance and scalability are dependent upon the application software architecture used in the benchmark.

This project aims to extend this approach to incorporate some predictive capability for applications running on COTS middleware products. This problem has a number of complex issues to address. For example, 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 executing 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, an empirical approach is economically impractical in a technological environment of rapid change and evolution.

9. References

- [1] W. Chiu, "Design Scalability-An Update", High Volume Web Sites, Software Group (AIM Division), September 2001
<http://www7b.boulder.ibm.com/wsdd/library/techarticles/hvws/scalability.html>
- [2] I. Gorton and A. Liu, "Software Component Quality Assessment in Practice: Successes and Practical Impediments", *the Proceeding of the International Conference on Software Engineering*, U.S., May 2002

- [3] A. Kang, "J2EE Clustering, Part 1", Javaworld <http://www.javaworld.com/javaworld/jw-02-2001/jw-0223-extremescale.html>
- [4] T. Jewell. "EJB 2 Clustering with Application Servers". http://www.onjava.com/pub/a/onjava/2000/12/15/ejb_clustering.html.
- [5] Seltzer, M., Krinsky, D., Smith, K., Zhang, X., "The Case for Application-Specific Benchmarking". *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII)*, March 1999, Rio Rico, AZ
- [6] Zhang, X., Seltzer, M. "HBench:Java: An Application-Specific Benchmarking Framework for Java Virtual Machines". *Proceedings of ACM Java Grande 2000 Conference*, June 3-5, 2000.
- [7] ECperf™ 1.0, Java Community Process™, <http://java.sun.com/j2ee/ecperf/>
- [8] Sun Microsystems. Enterprise JavaBean 2.0 Architecture. <http://java.sun.com/products/ejb/newspect.html>
- [9] Y. Liu, I. Gorton, A. Liu, N. Jiang, S.P. Chen, "Designing a Test Suite for Empirically-based Middleware Performance Prediction", *the proceedings of TOOLS Pacific 2002*, Sydney, Australia, Feb 2002
- [10] P. Jogalekar, C.M. Woodside, "Evaluating the Scalability of Distributed Systems", *IEEE Trans. on Parallel and Distributed Systems*, v 11 n 6 pp 589-603, June 2000.
- [11] M. Arlitt, D. Krishnamurthy, J. Rolia, "Characterizing The Scalability of A Large Web-based Shopping System", *ACM Transactions on Internet Technology (TOIT)*, Volume 1, Issue 1 (August 2001)
- [12] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", *IEEE Transactions on Computers*, Vol. 44, No. 1, January 1995, pp. 20-34.
- [13] J.Rolia, D.Krishnamurthy, and M.Litoiu "Performance Evaluation and Stress Testing for E-commerce System". *CASCON'98*, Canada
- [14] M.Litoiu, J.Rolia, and G.Serazzi, "Designing Process Replication and Threading Policies: a Quantitative Approach". *IEEE Transactions on software engineering*, Vol.26, No.12, December 2000.
- [15] E. Roman, S. W. Ambler, T. Jewell, F. Marinescu, *Mastering Enterprise EJB (2nd Edition)*, John Willy & Sons. ISBN 0471417114
- [16] E. Cocchet, J. Marguerite and W. Zwaenepoel, "Performance and scalability of EJB applications, submitted for publication" <http://www.cs.rice.edu/CS/Systems/DynaServer/index.html>, 2002
- [17] S. D. Kounev and A P. Buchmann, Performance Issues in E-Business Systems, SSGRR 2002w, Jan, 2002.