**The Association of System Performance Professionals**

The **Computer Measurement Group**, commonly called **CMG**, is a not for profit, worldwide organization of data processing professionals committed to the measurement and management of computer systems. CMG members are primarily concerned with performance evaluation of existing systems to maximize performance (eg. response time, throughput, etc.) and with capacity management where planned enhancements to existing systems or the design of new systems are evaluated to find the necessary resources required to provide adequate performance at a reasonable cost.

This paper was originally published in the Proceedings of the Computer Measurement Group's 2004 International Conference.

**For more information on CMG please visit http://www.cmg.org**

# Issues in developing a simulation model of an EJB system

David Mc Guinness, Liam Murphy, Andrew Lee
Performance Engineering Laboratory
Department of Computer Science
University College Dublin
Belfield, Dublin 4 Ireland

*Despite the fact that EJB (Enterprise Java Beans) is such a widely used technology, research in the area of performance modelling EJB application servers is quite sparse. This paper will describe how Ptolemy II, a discrete event simulator and general modelling tool, can be used to build a scalable model of an EJB system that allows users to input variables that describe interactions and their constituent methods, as well as system parameters. The model will output the average time for each given user interaction and allow users to seek system improvements by changing the system parameters and workloads.*

## 1 Introduction

The Java 2 Platform Enterprise Edition has demonstrated its use as an important standard for developing component-based multi-tier enterprise applications. The J2EE platform simplifies enterprise applications by basing them on standardized, modular components and providing a complete set of services to those components by handling many details of application behavior automatically, without complex programming [1]. A set of APIs is defined in J2EE to enable quick and efficient application building as well as a run-time infrastructure to host them. A key component of J2EE, Enterprise JavaBeans (EJB), has established itself as one of the leading architectures for developing and deploying secure, scalable and reliable applications.

As the use of such systems in Web-based, e-commerce environments expands, the need to model the performance of these systems becomes ever more important in facilitating optimal architectural design choices and system parameterisation. The goal of this paper is to describe a model that has been developed to simulate an EJB system in terms of its input parameters and to see what effect these inputs have on the response times for given interactions.

The modelling environment used is Ptolemy II. Ptolemy II is an open-source software for modelling, simulation and design of concurrent, real-time systems. It focuses on the use of well-defined models of computation that govern the interaction between components. A key principle in the Ptolemy project is the use of multiple models of computation in a hierarchical heterogeneous design environment. However our model operates within the confines of the Discrete Event (DE) domain, using also the Wireless domain to simplify the visual layout.

This paper continues as follows. Section 2 provides some background on EJB. Section 3 describes the Ptolemy simulation software, discusses the parameters that are input to the model and gives an outline of how the model runs and what its outputs are. Section 4 describes the experiments that were run and Section 5 draws some conclusions and mentions some future research directions.

## 2 EJB Environment

EJB defines a model for the deployment of reusable EJB components that can be assembled into a secure, scalable and reliable application that can run on any EJB-compliant application server.

The goal of this technology is to provide users with the following features:

- *Platform Independence* - Regardless of the underlying operating system, protocol or enterprise middleware services the EJB application will operate in a consistent manner.
- *Component Portability* - Components written for one EJB application can be reused in any other EJB product regardless of the vendor.
- *Standard, Agreed-Upon Technology* - that EJB is a common and standard technology means it is easier to find pre-trained staff, benefit from best-use principles, sell software and call upon external support.
- *Improved Development Efficiency* - The EJB architecture provides developers with a host of services (transaction, security, automated persistence, etc.) that enable the developer to quickly and efficiently create enterprise applications by focusing on application-specific

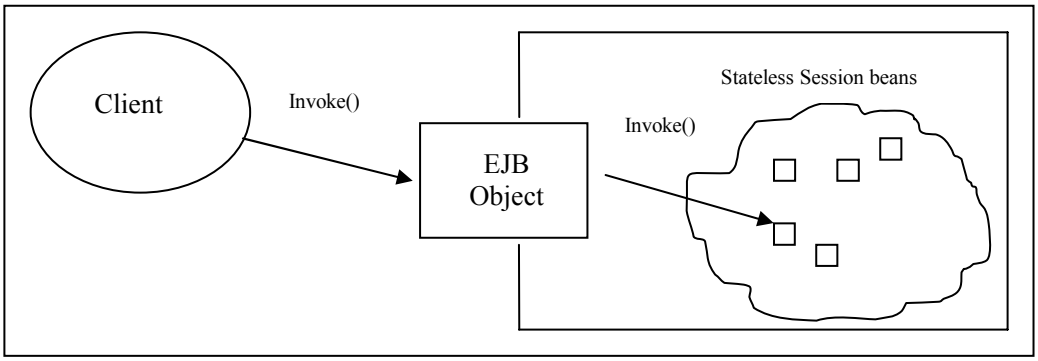


*Figure 1: Calling a Stateless Session Bean*

business logic [2].

### 2.1 EJB Architecture

The EJB architecture is designed to facilitate distributed, component-based computing. EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology. A key aim of this technology is to leverage Java's "*Write once, run anywhere*" model.

The EJB server offers various services including transactions (JTA), database management (JDBC), naming (JNDI), connection to legacy systems (JCA) and messaging (JMS). Each server contains at least one EJB container. The container is responsible for life cycle management, transaction control, persistence management, security, component pooling, authentication and access control. When the client invokes a server component, it is the responsibility of the container to automatically allocate a thread and a component instance on its behalf (see Figure 1). The container also manages all resources on the component's behalf and manages all interactions between the component and external systems.

### 2.2 EJB Components

The EJB Component Model comprises three different bean types alongside the EJB container that are used to carry out business interactions. These three bean types are Session Beans, Entity Beans and Message-Driven Beans (MD Beans). Session beans can be further sub-divided into stateless and stateful, whereas Entity Beans can be characterised by their method of persistence: Container-Managed Persistence (CMP) or Bean-Managed Persistence (BMP). Figure 2 represents these relationships graphically.

Session Beans typically represent business processes (such as a Credit Card Validator, a Pricing Engine, etc.) and are relatively short-lived entities. Session beans are further characterised by their conversational state. Stateful Session Beans contain conversational state on behalf of an individual client. They are intended to represent business processes that span multiple method requests (e.g. a shopping cart). Stateless session beans on the other hand represent single request conversations and as such do not retain state. Because of this stateless session beans can easily be pooled and reused by several different clients. An example of the use of a stateless Session Bean could be a Credit Card Validator (that receives the credit card details as an input parameter) but doesn't need to hold any state beyond the method execution.

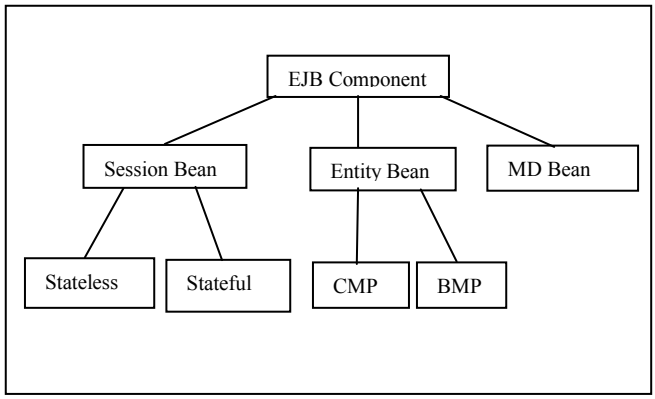


*Figure 2: The triad of EJBs*

Entity Beans, unlike Session Beans, are persistent objects that can be stored in permanent storage. Entity Beans typically represent business data and usually map to a line of a database table. An example of such business data could be a client's bank account. Entity Beans can have their persistence carried out for them by the container (CMP) or within the bean itself (BMP) by means of hand-coded JDBC statements.

Message-Driven (MD) Beans are beans that use the lightweight communication known as messaging, instead of RMI-IIOP (Remote Method Invocation over Internet Inter-Orb Protocol). A middleman sits between the client and the server and deals with forwarding the requests. One of the major advantages of these beans is that unlike RMI-IIOP they do not have to block to wait for a response, enabling processing to continue.

It has been shown [5] that stateless session

beans with BMP can run very efficiently (comparable to servlet-only implementations). Hence our model is designed to represent only stateless session beans at this time.

# 3 Model Design

## 3.1 Ptolemy

The modelling environment used is Ptolemy. Ptolemy is an open-source software for modelling, simulation and design of concurrent, real-time systems developed at Berkley University [2]. It focuses on the use of well-defined models of computation that govern the interaction between components. A key principle in the Ptolemy project is the use of multiple models of computation in a hierarchical heterogeneous design environment.

The discrete-event (DE) domain supports time-oriented models of systems such as queuing systems, communication networks, and digital hardware. In this domain, actors communicate by sending *events*, where an event is a data value (a token) and a *time stamp*. A DE scheduler ensures that events are processed chronologically according to this time stamp by firing those actors whose available input events are the oldest (having the earliest time stamp of all pending events). The fact that simultaneous events are handled systematically and deterministically ensures a high level of dependability in models constructed with Ptolemy. Additionally an efficient structure for global event queues ensures that the overhead for this list does not become prohibitive [3].

## 3.2 Parameterization

There are many parameters that affect the performance of an EJB server and where possible these parameters have been included in the model. The model also uses several *background* parameters that are also mentioned to aid understanding how the model works.

### 3.2.1 User-modifiable parameters

The parameters detailed in Table 1 are input parameters to the model which can be modified by the user before running the model. All of these parameters are set at runtime and remain constant throughout the model execution.

### 3.2.2 Model Parameters

Additional runtime parameters used by the model but not modifiable by the user are shown in Table 2.

| **User-modifiable parameters** |
|---|
| *numberOfThreadsAvailable* |
| *memorySize* |
| *numberOfBeansDeployed* (array) – each bean's deployment is detailed here |
| *numberOfCPUs* |
| *CPUSpeed* (scalar) |
| *I/OSpeed* (scalar) |
| *Bandwidth* (scalar) |
| *numberOfUsersPerInteraction* (array) |
| *endTime* (when the model should stop executing) |
| Interaction Parameters (array of arrays) – each interaction is represented by a single array. This array itself contains the following parameters:<br>➢ *CPUTimeRequired,*<br>➢ *memoryUsage,*<br>➢ *IOTimeRequired,*<br>➢ *beanTypeRequired,*<br>➢ *localOrRemoteCall* (0 for local / 1 for remote / -1 denotes end of interaction) |

*Table 1: User-modifiable Parameters*

| **Model Parameters** |
|---|
| *numberOfThreadsInUse* |
| *memoryInUse* |
| *beanCounts* (array) |
| *warmUpPeriod* |
| *Event Token Parameters* – each Event Token contains 4 parameters:<br>➢ *Timestamp*<br>➢ *interactionIdentifier*<br>➢ *indexForInteractionArray*<br>➢ *totalMemoryInUseForInteraction* |

*Table 2: Model Parameters*

These parameters are all modified by the system as it executes and represent the state of the system (except *warmUpPeriod*, *Timestamp* and *interactionIdentifier* whose values do not change). Event Token Parameters represent the state of a currently-executing interaction.

## 3.3 Model Design

Some previous work exists that looks at the performance of EJB Systems (see [4], [5], [6]). There are also some analytical models of an EJB Server (see [7], [8]). However there is little or no work in the area of simulation of EJB systems, particularly ones that may be used by those with little simulation experience. Due to the accuracy that can be achieved with simulation models in comparison with analytical models, as well as the advantages in flexibility of simulations compared to real working systems, it is felt that a simulation model of an EJB system is overdue. Simulation models have many advantages over analytical ones such as the ability

to model the effects of priority scheduling as well as transient and bursty effects and the ability to compute complex statistics such as the 95th percentile.

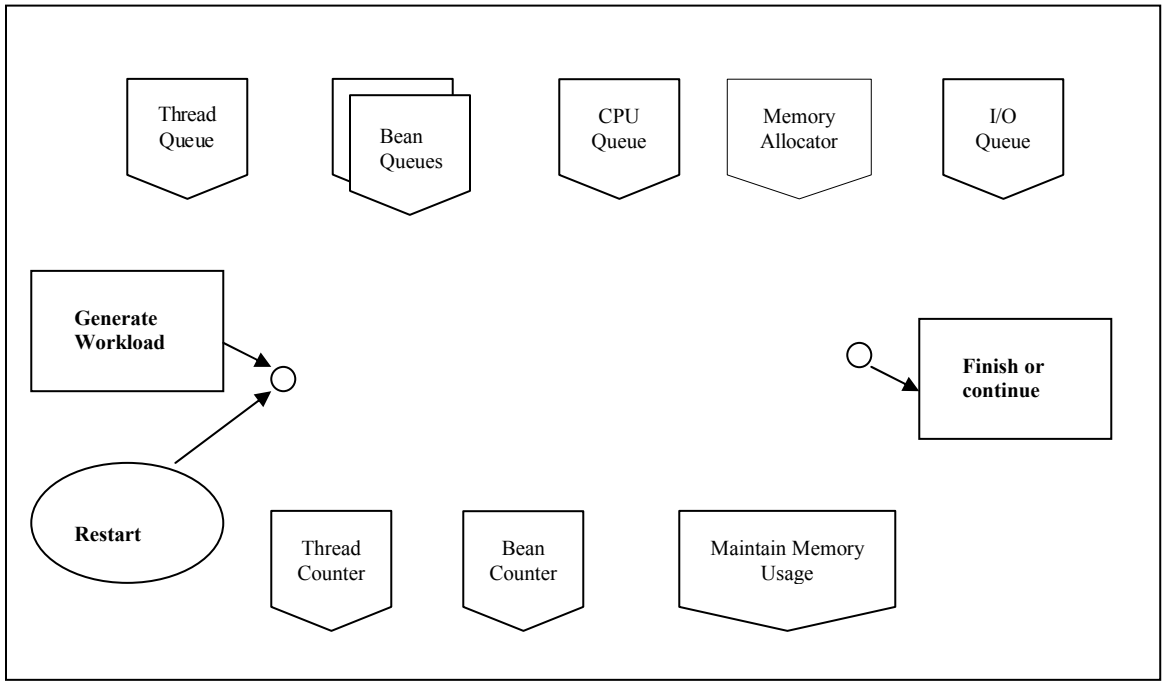


*Figure 3: Model of Ptolemy with Submodels*

In Web-based EJB systems it is the Web-server or more specifically, proxy client components created by the web-server that call the EJB server. As our area of interest is purely the EJB server it was decided that it would be modelled as a closed system.

Many simplifications of a real system have been made in order to ensure that the model doesn't become too cumbersome to use or too expensive in terms of execution time. There is a fine balance between the amount of detail necessary to make a model meaningful and a level of detail that is very CPU-intensive and hence runs are restrictively long [9]. The simplifications for this model include the facts that the CPU is visited only once per method, that all CPUs on the machine have equal power, that all methods have equal priority, and that security has not been modelled.

At present the model represents a single server but work is under way to create a multi-server version of the model, where parameters such as *numberOfCPUs*, *numberOfThreadsAvailable* etc. will be configurable separately for each server. This model may require further simplification to prevent simulations becoming too time-consuming. Elements of the model that are seen to have only a small impact on output values may be simplified or ignored. As can be seen in Figure 3 the current model consists of several submodels named: *GenerateWorkload, ThreadQueue, BeanQueues, CPUQueue, MemoryAllocator, I/OQueue, MaintainThreadCount, MaintainBeanCounts, MaintainMemoryUsage* and 2 submodels (the *finishOrContinue* submodel and the *Restart* submodel) to check if an interaction is completed. An interaction is completed if:

*interaction.length = indexForInteractionArray*

and so re-enter the system at the beginning or needs to execute more methods, if:

*interaction.length > indexForInteractionArray*.

The *finishOrContinue* submodel checks to see if the current interaction has performed all of its methods. If it has not, it continues to execute its methods to the end. If it has, it enters the Restart submodel where statistics are taken on end-to-end interaction time for those interactions that started after the warm-up period completed (i.e. when the full population of the model had entered the system).

The *GenerateWorkload* Model creates an Event Token for each user in the system (taken from the *numberOfUsersPerInteraction* array) with an *interactionIdentifier* to keep track of it. Entity Tokens are released into the system at a fixed rate throughout the warm-up period:

release rate = (warm-up period)/(number of users for the given interaction)

As each Event Token is released into the system a time-stamp is attached to it which ensures that its completion

time can be calculated and allows it to be traced in the system. The *indexForInteractionArray* and the *totalMemoryInUseForInteraction* parameter are set to zero at the start of each interaction and are modified as necessary as they move through the model. See 3.2.2 for details of the parameters contained by an Event Token.

The Event Tokens are moved throughout the system in this general order:

*ThreadQueue-BeanQueue-CPUQueue-MemoryAllocator-I/OQueue-FinishOrContinue*.

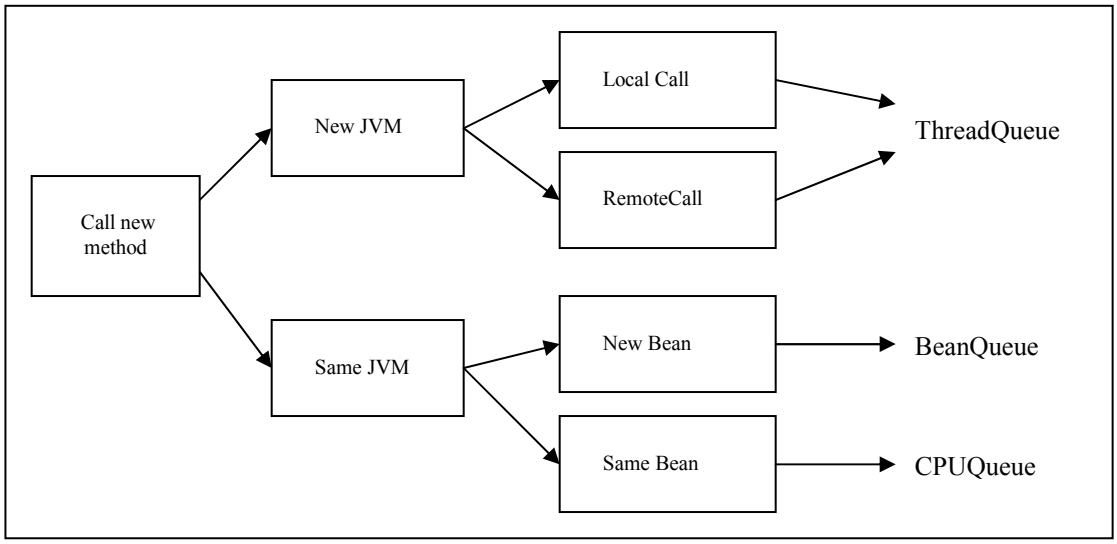


*Figure 4: Routing in the EJB Model*

Each method may follow some variation of this path (for example if there is no I/O associated with a particular method then it will not enter the I/OQueue or if a method other than the first in the interaction continues to another method within the same JVM and requires a local call it will not enter the *ThreadQueue,* nor the *BeanQueue* if it needs the same bean again). See Figure 4.

Each of the first 3 submodels at the bottom of Figure 3 (*ThreadCounter, BeanCounter* and Maintain*MemoryUsage*) work in connection with their equivalent queues at the top (*ThreadQueue, BeanQueue* and *MemoryAllocator* respectively). For example when an Event Token enters the *ThreadQueue* submodel and reaches the front of the queue, it checks to see if there is an available thread:

*numberOfThreadsAvailable* > *numberOfThreadsInUse*

If not, it waits in the queue for an available thread. If there is, it proceeds and sends a message to the *MaintainThreadCount* submodel to increment the *numberOfThreadsInUse* parameter. Conversely when an interaction is finished with a thread (when an interaction is finished or when a remote call is necessary) a message is sent to the *MaintainThreadCount* submodel to decrement the *numberOfThreadsInUse* parameter. This submodel then sends a message to the *ThreadQueue* submodel to release the next Event Token in the queue awaiting a thread, if there is one, and the process starts over.

The *MemoryQueue* submodel has a similar relationship with the *MaintainMemoryUsage* Submodel except that in the queue it needs to check if:

*memoryInUsage* + *"current memory requested"* < *memorySize*

before letting the Event Token proceed. Additionally the *MaintainMemoryUsage* submodel needs to increase the amount of *memoryInUsage* by the amount of memory allocated or released. Accumulated memory for a particular interaction is stored as one of the parameters attached to the Event Token. Hence each method's memory requirement in the Interaction array is the net increase in memory needed for that method and when the interaction completes, the total amount of memory accumulated by the interaction is released.

The *BeanQueue* submodel is more or less the same model as the *ThreadQueue* submodel except that it needs to do the same for each particular bean type. This is achieved in Ptolemy by using the *MultiCompositeActor* (a built-in Ptolemy Actor) that creates multiple instances of the same submodel at runtime. The number of instances is determined by the number of types of beans deployed in this case. Each particular bean request is then routed to its appropriate submodel and hence to its appropriate queue. The queue then checks that the bean in question is available before letting the Event Token proceed by checking the appropriate element in each of the following arrays:

*beansInUse[beanTypeRequired] < NumberOfBeansDeployed[beanTypeRequired]*

The *MaintainBeanCount* submodel then increments the appropriate element in the *NumberOfBeansInUse* array. A similar situation arises when a bean is released as occurs in the *MaintainThreadCount* submodel (except the specific bean number count is decremented). Beans as well as submodels created at runtime by the *MultiCompositeActor* are numbered 0, 1, 2, etc. and so routing is done transparently by use of the appropriate parameter in each case.

The *CPUQueue* submodel implements a queue without use of the Server actor in Ptolemy. This is achieved in a manner similar to a semaphore. The *Number of CPUs* determines the number of CPU Tokens available. When an Event Token arrives at the *CPUSubmodel* it looks to see if there is an available CPU Token. If not it waits for one before proceeding. If there is, it reads the *CPUTime* parameter associated with the method it is currently executing and holds the CPU Token (and the Event Token) for this length of time. When it is finished executing it passes on the CPU Token to the next Event Token in the queue (if there is one) and proceeds.

The *IOQueue* works essentially as a single Server actor that services each Event Token in the order it receives them (one at a time) and releases them after an amount of time determined by the *IOTimeRequired* parameter in the current Interaction (see Interaction Parameters under 3.2). An *IOScalar* parameter is also used to be able to show the effects of changing the underlying I/O hardware, e.g. if the new I/O hardware is 20% faster, you can set the *IOScalar* parameter to 1.2 to indicate this change. This could also be used to reflect changes in the efficiency of JDBC code inside the stateless session beans.

### 3.4 Model Outputs

The model collects various statistics that can be used to collect various pieces of information about the system. As previously mentioned, only those statistics that refer to events that were started after the warm-up period will be included so as not to include details of events that execute before the system is fully populated (as these would execute interactions faster than the steady-state interactions, as well as cause resource utilisation statistics for the steady-state to be undervalued).
Outputs include:

- *averageInteractionExecutionTime (*end-to-end*) for each interaction*
- *throughputPerInteraction*
- *averageUtilisationPerCPU*
- *maximumBeansInUse (*for each type*)*
- *maximumThreadsInUse*
- *IOUtilisation*
- *maximumMemoryInUse*

## 4 Experiments

The goal of the current set of experiments is to see how much of an impact input parameters have on specific output parameters. For example:

- does increasing the number of processors by 20% have more or less of an impact on *averageCPUUtilisation* and the individual *interactionTimes* than increasing the power of the existing CPUs by 20%?
- or how is the system impacted when the CPU/IO are reaching their saturation levels?

| Parameters | Int. 1 | Int. 2 | Int. 2 |
|---|---|---|---|
| CPUTimeRequired | 0.5 | 0.3 | 0.65 |
| memoryUsage | 8750 | 10000 | 1000 |
| IOTimeRequired | 0.025 | 0.125 | 0.175 |
| beanTypeRequired | 1 | 2 | 4 |
| localOrRemoteCall | -1 | 1 | 0 |
| CPUTimeRequired | | 0.1 | 0.25 |
| memoryUsage | | 9500 | 1870 |
| IOTimeRequired | | 0.15395 | 0 |
| beanTypeRequired | | 3 | 4 |
| localOrRemoteCall | | -1 | -1 |

*Table 3: Interaction Values Used in Experiments*

Table 3 shows the values used in the experiments we carried out. These values remained constant across all experiments. The first experiment examines the effect of increasing the number of CPUs on the average completion time for each interaction as well as the throughput for each. It looks at the situation where CPU utilisation is at 100% both before and after the changes are made (i.e. the CPU is the bottleneck).

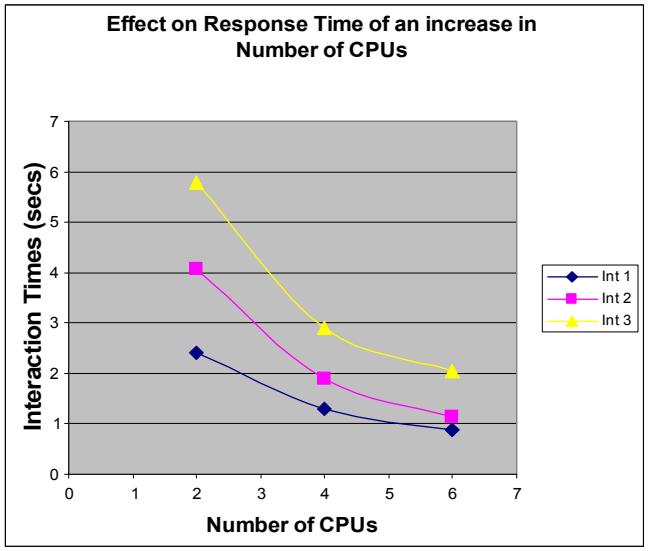


*Figure 5a: Number of CPUs effect on response times*

The relationship in this case between *numberOfCPUs* and *interactionTime* appears to be asymptotic (see Figure 5a), whereas the relationship between *numberOfCPUs* and throughput is seen to be almost linear (see Figure 5b). Notice that there is a diminishing of returns in terms of response time when CPUs are added. Interestingly there is also a linear relationship between the *numberOfCPUs* and the utilisation of I/O devices (Figure 5c). This is the case as long as the usage of the I/O device is not near maximum usage but the CPU usage is at, or close to, 100%.

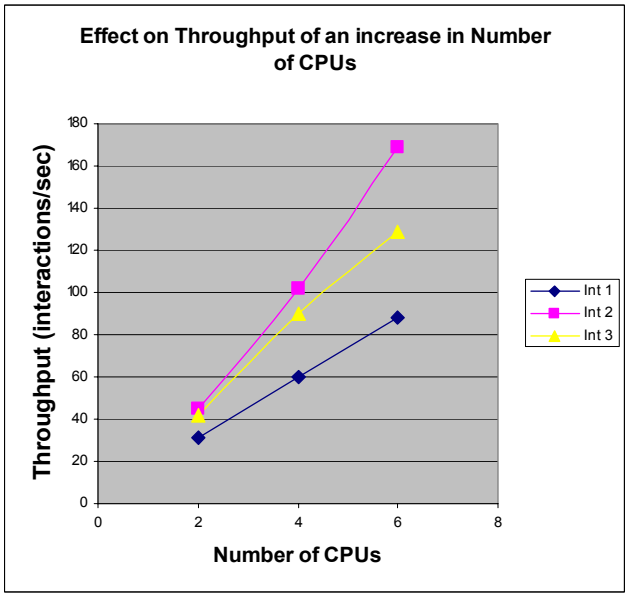


*Figure 5b: Number of CPUs effect on throughput*

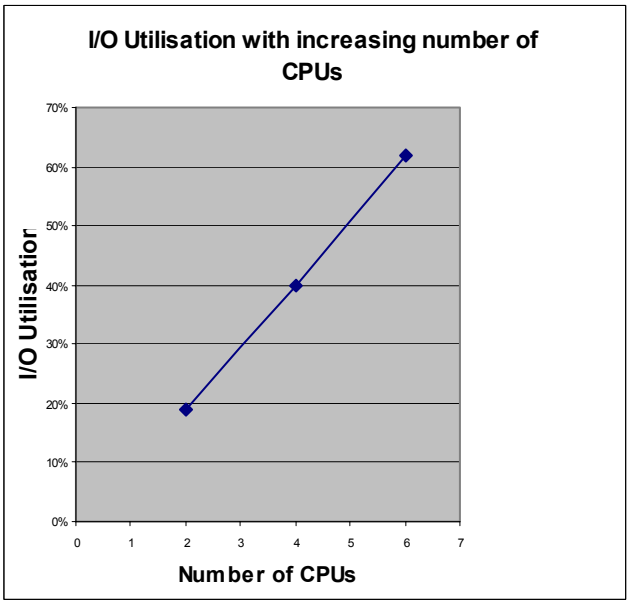


*Figure 5c: Number of CPUs effect on I/O Utilisation*

The second experiment looks again at the *numberOfCPUs* but this time it looks at comparing an increase in the *numberOfCPUs* to an equivalent increase in the power of existing CPUs. It can be seen that the latter case (as may have been expected) shows reasonably similar behaviour to the former (see Figure 6). The first case is 2 CPUs with a power rating of 1. The second case is 2 CPUs, each with a power rating of 2, and the third case has 4 CPUs, each with power rating of 1. (A CPU with power rating 2 executes twice as fast as one with power rating 1).

As the CPU was the bottleneck in each of these experiments, it was again I/O Saturation that changed as the *numberOfCPUs* or the *CPUscalar* changed (see Figure 6c), and the relationship is again linear (as long as the CPU is the bottleneck before and after the changes). In some interactions the improvement was greater to both interaction time and throughput by increasing the power of the CPUs (as opposed to increasing the number), however this only held for some particular interactions and the response time and throughput for other interactions was worse.
In all, there were found to be no major advantages to be gained for this particular set of data by increasing CPU power over number of CPUs or vice versa.

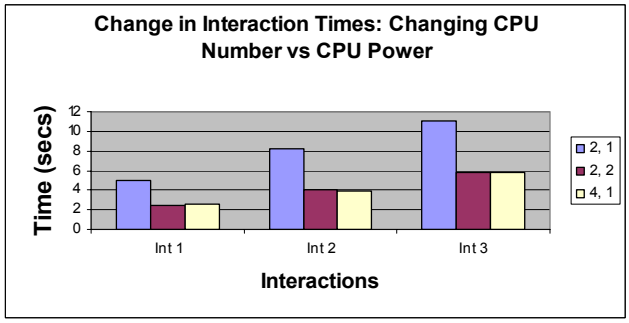


*Figure 6a) Effect on Interaction Times of CPU Strategies*

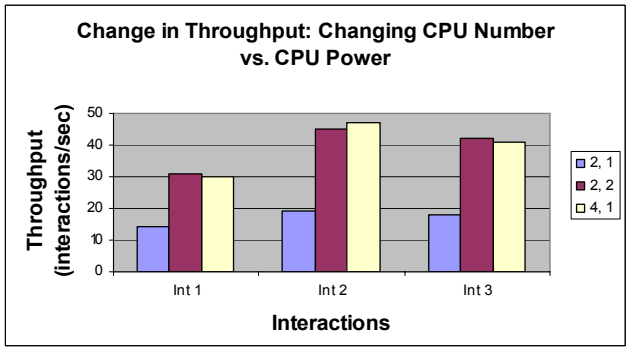


*Figure 6b) Effect on Throughput of CPU Strategies*

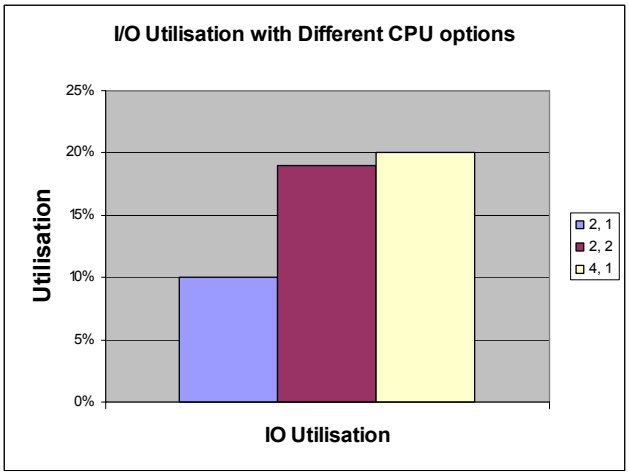


*Figure 6c) I/O Utilisation with different CPU Strategies*

The third experiment looks at the effect of the size of the thread pool both on interaction times and also on the throughput for each of the interactions.

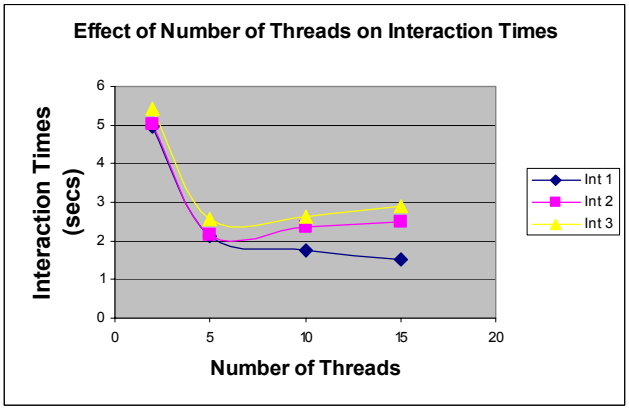


*Figure 7a: Number of Threads effect on Interaction Times*

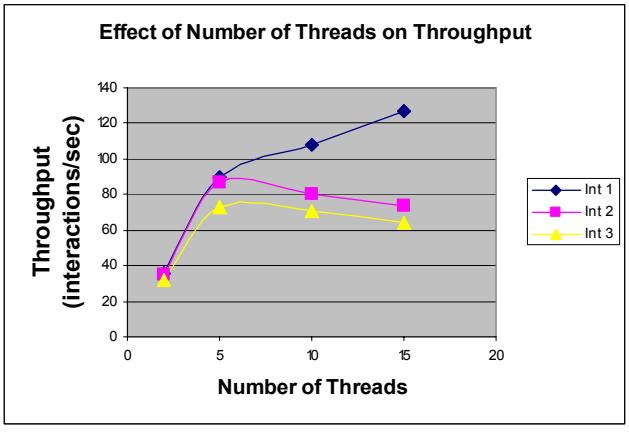


*Figure 7b: Number of Threads effect on Throughput*

Again what is seen to be of key importance is the idea of a bottleneck device. If the CPU is not at maximum usage then the effect of changing the number of threads available is less significant than when it is at maximum utilisation. In this example, with its particular set of input data, it can be seen that the optimal number of threads to have available is about 5 or 6 as in each of the graphs this is where the 'knee' of the curve is. Increasing the number of threads beyond 5 brings quickly diminishing returns in terms of effects on Throughput, Interaction Times or Resource utilisation, whereas, decreasing it below 5 will show a sharp degradation in each of these measures. This shows how the model can quickly be used to determine the optimal number of threads that should be available for a particular workload and a given hardware configuration.

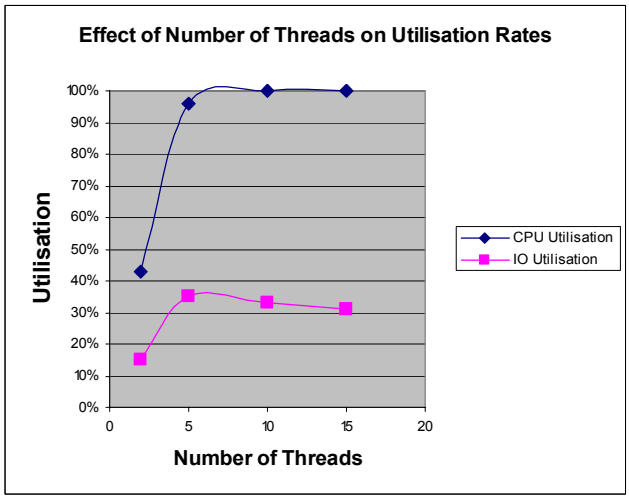


*Figure 7c: Number of Threads effect on Resource Usage*

The fourth experiment looks at the effect of changes in the workload on response times (interaction times) and on throughput. The workload is simplified in each case so that the number of users of each interaction is the same for each test, to examine only changes in the size but not the mix of the workload. Therefore the number of Users was the same for each interaction in any particular test carried out. Again the effects of the changes in the workload were heavily dependent on the level of saturation of the system.

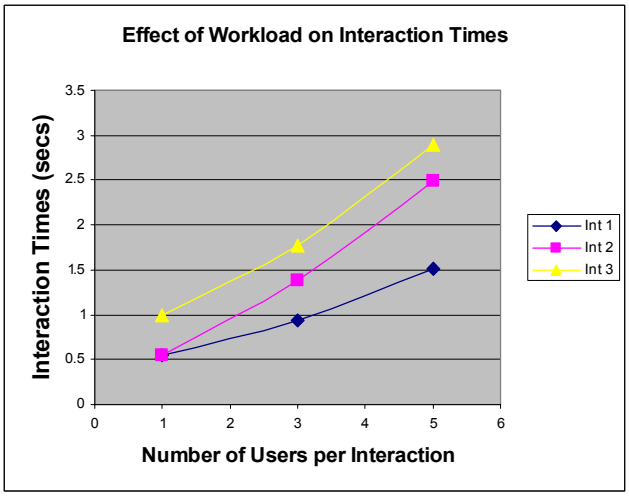


*Figure 8a) Workload effect on Interaction Times*

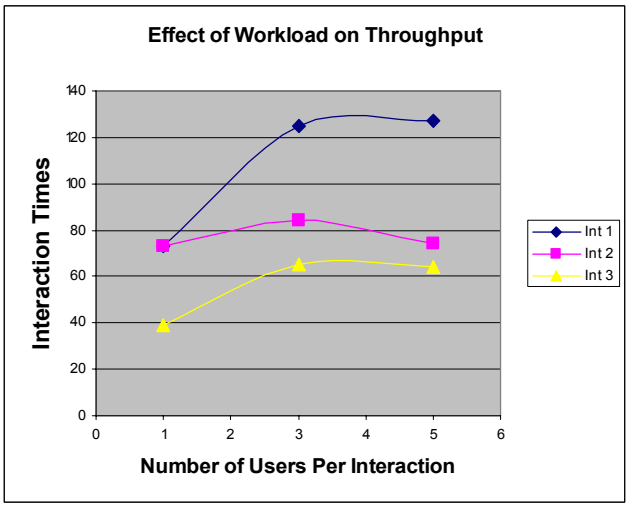


*Figure 8b) Workload effect on Throughput*

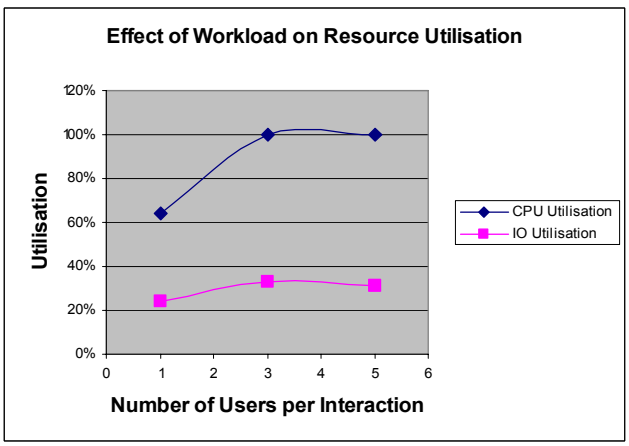


*Figure 8c) I/O Utilisation under varying Workloads*

The higher the level of saturation of the system, the greater the impact of changes in the workload, both on interaction times and on throughput. Looking at the Resource Utilisation Figure reveals that at 3 Users per interaction the CPUs are already stretched to full utilisation. Therefore at this stage the increase in Interaction Times becomes more marked and the gains in throughput that were experienced up to that point by increasing the workload begin to fall away and overall throughput begins to fall.

The fifth experiment looks at the effect of bean deployment on interaction times and on throughput. Using the same logic as we did with number of users in the previous experiment, we have deployed the same number of each bean to simplify matters.

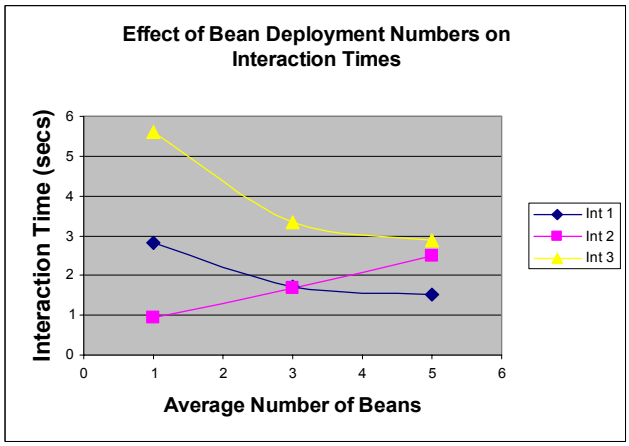


*Figure 9a: Effect of the Number of Beans on Interaction Times*

It may have been expected that an increase in the number of beans deployed would improve the throughput as well as decrease the interaction times.

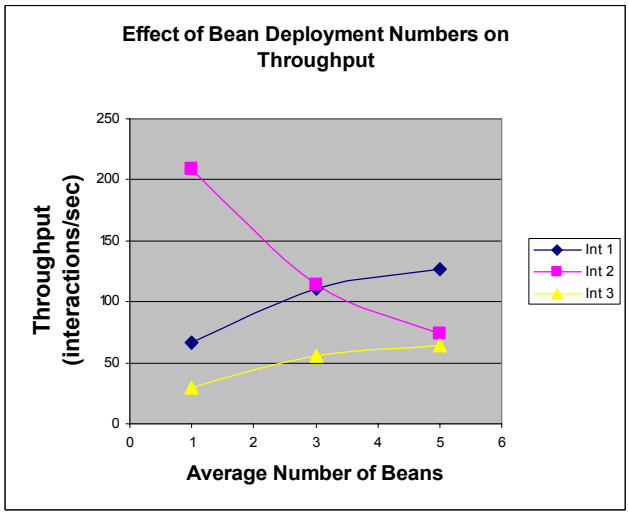


*Figure 9b: Effect of the Number of Beans on Throughput*

This is true for two of the three interactions. Interaction 2 however works in precisely the opposite way we may have expected. As the number of beans deployed increases the throughput decreases and the response time for this interaction increases. This may serve as a reminder that systems do not always behave as we might expect and this demonstrates the need for simulation models to catch any possible deviations from expected behaviour before physical changes are made

to systems.

The next step is thus to verify and validate that this is a real system artefact and ensure it is not a bug in the model.

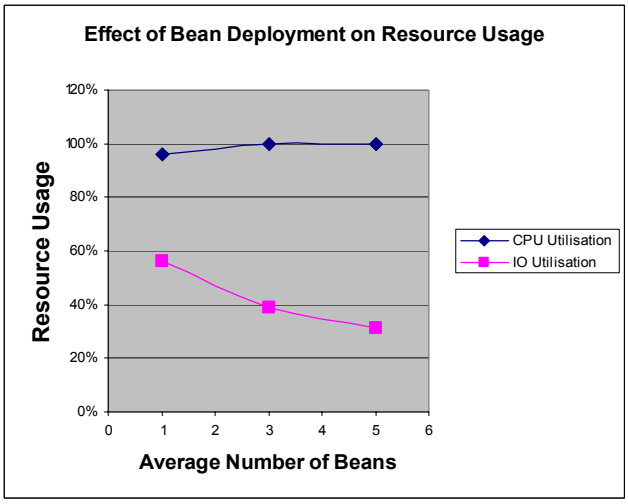


*Figure 9c: Effect of the Number of Beans on Resource Usage*

## 5 Conclusions

For EJB Systems like many other large systems it is very difficult to estimate performance given the profusion of factors that influence its performance. For this reason a simulation model that represents the most important elements in the system can aid in making better architectural choices as well as help reveal trade-offs and difficulties in trying to guarantee performance in Enterprise-level EJB systems. With multiple interactions, beans, CPUs, users and so on, it is very difficult to generalise about the effects of many of the changes. Much depends on the bottleneck and if the bottleneck of the system is not fully understood the effects of hardware or software changes can cause some unexpected results, even in a simulation environment where parameters such as *numberOfUsers* can be held constant. Not only can this model be used to predict performance indicators for EJB systems, it can also be used to build intuition in users of how changes to a real EJB server would be likely to affect individual interaction (response) times and throughput and helps the user to ascertain which change's effects can more easily be generalised in terms of performance, and which need more careful analysis. There are many simplifications in the model to prevent it being prohibitively expensive in terms of running time but it now needs to be verified that the model can still perform accurate tests in terms of real running EJB systems and to see if the simplifications made were reasonable ones or if certain levels of detail that are included in the model are redundant. Future work should also look at creating a multi-server version of the model used as a basis for this paper. The model would then need to be verified against a distributed, enterprise-level EJB system.

## 6 Acknowledgements

The support of the Informatics Research Initiative of Enterprise Ireland is gratefully acknowledged.

## 7 References

[1] http://java.sun.com/products/ejb/

[2] http://ptolemy.eecs.berkeley.edu/ptolemyII

[3] http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII3.0/ptII3.0.2/doc/design/ptIIdesign2-software.pdf (Actor Package)

[4] S. Ran, P. Brebner, I Gorton, "*The Rigorous Evaluation of Enterprise Java Bean Technology",* a long paper in the 15th International Conference on Information Networking (ICOIN-15), 2000.

[5] Ian Gorton, Anna Liu, "*Evaluating the Performance of EJB Components*". IEEE Internet Computing 7(3): 18-23 (2003).

[6] Emmanuel Cecchet, Julie Marguerite, Willy Zwaenepoel, "*Performance and Scalability of EJB Applications".* (OOPSLA'02).

[7] C. M. Lladó, J. Lüthi, P. G. Harrison, *"Studying Sensitivities of an EJB Performance Model"* (MASCOTS'02).

[8] Samuel Kounev, Alejandro Buchmann, *"Performance Modeling and Evaluation of Large-Scale J2EE Applications"* (CMG'03).

[9] D. J. Lilja – *"Measuring Computer Performance"* 2000, ISBN: 0-521-64105-5.