Performance Evaluation of an Enterprise JavaBean Server Implementation

Catalina M. Lladó*
Department of Computing, Imperial College of Science, Technology and Medicine
180 Queens Gate
London SW7 2BZ, United Kingdom
cl16@doc.ic.ac.uk

Peter G. Harrison
Department of Computing, Imperial College of Science, Technology and Medicine
180 Queens Gate
London SW7 2BZ, United Kingdom
pgh@doc.ic.ac.uk

ABSTRACT

An analytical performance model is developed for the server in a three-tier, client-server system running in an object-oriented environment. The server implements Sun's EJB architecture and its method execution mechanism is abstracted as a queueing network with a non-standard form of blocking. This queueing network is then reduced by aggregating subnetworks into single nodes with queue length dependent service rates that capture the essence of the blocking effects. The analytical model of the server's method calling processes is then validated against simulation in terms of both throughput and queue length distribution, good agreement being obtained in both cases. Finally, analytical predictions for throughput in the whole system model are compared with simulated ones.

1. INTRODUCTION

The increasing complexity of the new object-oriented software technologies leads to the necessity for some form of quantitative (performance) model when a system is designed. This is particularly true when the system itself is a complex, distributed, client-server system. In this paper, we develop an analytical model for the central scheduler of a distributed, three-tier, client-server architecture, typical for large, Java-supported, Internet applications. The server implements the EJB-1.1 (*Enterprise JavaBeans*) specification [9] which is a new component architecture, created by Sun, for the development and deployment of object-oriented, distributed, enterprise-level applications.

Concretely, we investigate one of the server functionalities – the entity method execution call, defined in section 3. The objective is to analytically model the method call execution process in order to predict its performance in terms of throughput (number of method executions per unit time) and mean response time. This is achieved using a queueing network model with certain non-standard features arising from a form of blocking. Following this, the same process is simulated and the results thus obtained used to validate the analytical model. The queueing model is then integrated as a flow equivalent server (FES) into a system model that describes the dynamics and performance of threads cooperating in an object-oriented environment, viz. the EJB server implementation.

The rest of this paper is organised as follows. Section 2 describes the EJB architecture and the implementation. Section 3 explains the entity method call execution process and its analytical model is developed in section 4, along with the system model. Numerical results are presented in section 5 where the method call execution model is validated against simulation. The paper concludes in section 6, where future directions for investigation are also discussed.

*Catalina Lladó's research is partially funded by "Sa Nostra, Caixa de Balears".

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

WOSP 2000, Ontario, Canada

© ACM 2000 1-58113-195-X/00/09 ...$5.00

2. ENTERPRISE JAVABEANS

Enterprise JavaBeans is an architecture for component-based distributed computing. Enterprise Beans are components of distributed transaction-oriented enterprise applications. Each component (or bean) lives in a container. Transparently to the application developer, the container provides security, concurrency, transactions and swapping to secondary storage for the component. Since a bean instance is created and managed at runtime by a container, a client can only access a bean instance through its container.

EJB defines two kinds of components: *session* beans and *entity* beans. While *session* beans are lightweight, relatively short lived, do not survive server crashes and execute on behalf of a single client, *entity* beans are robust, expected to exist for a considerable amount of time, do survive server crashes and are shared between different users.

A *bean implementation* and two interfaces, the *bean interface* and the *home interface*, define a bean class. A client accesses a bean through the bean interface, which defines the business methods that are callable by clients. The home interface allows the client to create and, in case of entity beans, look up a bean. From the bean implementation and

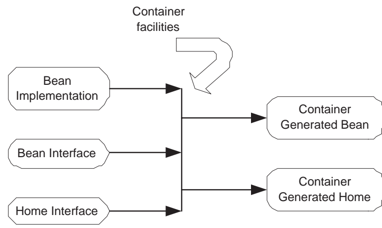

Figure 1: Class generation using container facilities

the interfaces, two new classes are automatically generated using the container facilities: a bean class that will wrap the actual bean (*Container Generated Bean* or CGBean class) and a home class that allows a user to create a bean (*Container Generated Home* or CGHome class). Fig. 1 shows the class structure.

In the EJB implementation under study, each container is responsible for managing one EJB. Thus, there is one container instance (from now on simply referred to as a container) for each bean class. Fig. 2 shows how a client uses different bean instances of the same bean class.

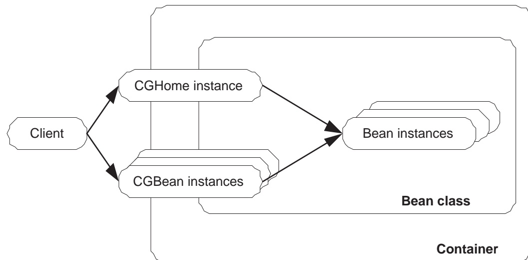

Figure 2: Client use of several bean instances

The communication between clients and EJB server is done through *Remote Method Invocations* (RMI), which is a protocol allowing Java objects to communicate [8]. A method dispatched by the RMI runtime system to a server may or may not execute in a new, separate thread. Some calls originating from the same client virtual machine will execute in the same thread and some will execute in different threads. However, calls originating from different client virtual machines always execute in different threads. Therefore there is at least one thread for each client virtual machine.

The EJB Server and container implementations use *Java synchronised* statements and methods to guarantee consistency when objects (i.e. bean instances and containers) are concurrently accessed. Due to the fact that synchronised statements and methods have an important influence on system performance, their detailed behaviour is explained below. Similarly, the characteristics and behaviour of entity beans are also described in more detail after this.

2.1 Java Synchronisation

In order to synchronise threads, Java uses monitors [7], which are high-level mechanisms allowing only one thread at a time to execute a region of code protected by the monitor. The behaviour of monitors is explained in terms of locks; there is a lock associated with each object. There are two different types of synchronisation: *synchronised statements* and *synchronised methods.*

A synchronised statement performs two special actions:

1. After computing a reference to an object, but before executing its body, it sets a lock associated with the object.
2. After execution of the body has completed, either normally or abortively, it unlocks that same lock.

A synchronised method automatically performs a lock action when it is invoked. Its body is not executed until the lock action has successfully completed. If the method is an instance method, the lock is associated with the object for which it was invoked (that is, the object that will be known as *this* during execution of the body of the method). For a class method, i.e. if the method is static, the lock is associated with the class object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abortively, an unlock action is automatically performed on the acquired lock.

2.2 Entity Bean Implementation

An entity container has a limited number of bean instances that can exist concurrently. In other words, there is a maximum number of active bean instances (i.e. instances in main memory ready to be accessed) for a bean class.

Multiple clients can access an entity instance concurrently. In this case, the container synchronises its access. Each client uses a different instance of the CGBean class that interfaces the client with the container but all the clients share the same bean instance. As shown in Fig. 3, when two clients access an instance concurrently, they share the CGHome instance (because there is only one of them for all the clients) but they use different CGBean instances.

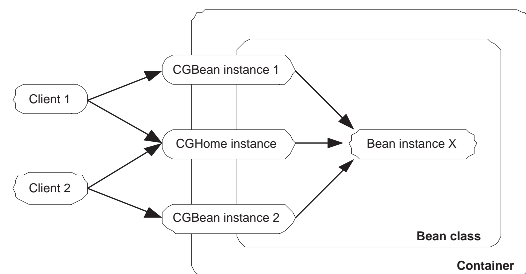

Figure 3: Concurrent access from two clients

3. METHOD CALL EXECUTION

Method invocations are made through the container in order to perform transparent user authentication and to allocate the necessary resources to execute the methods. As stated already, bean instances of the same bean class share the container. As a consequence, threads (or clients) using either the CGHome or CGBean instances of the same bean class share the lock for its container. The lock is requested when these objects invoke a container synchronised method since

only one of the synchronised methods for each container can be executed at the same time, there being only one class per container. In addition, CGBean instances use the thread manager which also has synchronised methods, in particular StartMethod and FinishMethod as indicated in Fig. 4. Therefore, all the CGBean instances share a resource, which is the thread manager lock.

There is a limit to the number of method calls that can be concurrently executing. Hence, when the method call number reaches this maximum, the CGBean requesting a thread to carry out a method execution will wait until one of the executing methods finishes. Similarly, clients who share a bean instance share the lock associated with it when they use synchronised statements. The interaction diagram [1] for a method execution call is shown in Fig. 4.

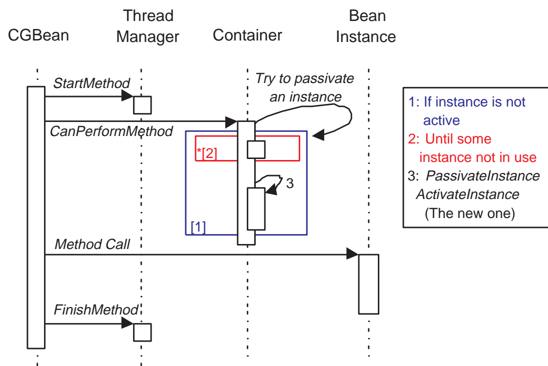

Figure 4: Interaction diagram for a Method Execution Call

As a result of the synchronisation at various levels – the thread manager, containers and synchronised method execution, threads must queue first-come-first-served for the resources at each level. An appropriate modelling formalism is therefore a queueing network, with various non-standard features, as we discuss in the next section.

4. AN EXTENDED QUEUEING MODEL

Based on the behaviour explained above, a queueing network model is shown in Fig. 5. The queueing network consists of $1 + C + C * M$ stations, where 1 corresponds to the thread manager station, C is the number of containers in the system (i.e. the number of different bean classes) and M is the maximum number of (different) bean instances for a bean class that can be active at the same time. Since we are modelling the performance of synchronised methods, every node in the model has first-come-first-served (FCFS) queueing discipline. Unsynchronised methods could be incorporated either as an overhead on each instance-server node or via additional parallel instance servers with processor sharing (PS) queueing discipline. Each node in the model represents a process which may or may not be implemented on its own processor. We assume the former but it would be relatively straightforward to model multiple processes executing on single (or parallel) servers using a process-sharing queueing discipline to determine the queue length dependent rate of the aggregate server, cf. the methods used in the rest of this section.

The particular set of active instances for any class, associated with a bean container and executing on up to M servers in the model, will vary over time according to the demands of the tasks departing from the container server. We assume that switch-over time between active instances at the parallel servers is negligible and accounted for in the container's service times. The 'waiting set' (refer Fig. 5) comprises threads not admitted to the whole EJB server network which also has a concurrency limit, N. As in traditional multi-access system models (see, for example, [5]), we solve for the performance of the whole *closed* queueing network, with the waiting set and all departures removed, at various populations N.

For mathematical tractability and the desire for an efficient (approximate) solution, we assume all service times are exponential random variables, that the queueing disciplines are FCFS and that routing probabilities are constant and equal across each of the C bean containers. The M bean instances attached to each bean container are also equally utilised overall, but the specific routing probabilities in each network-state depend on the blocking properties, which are described below.

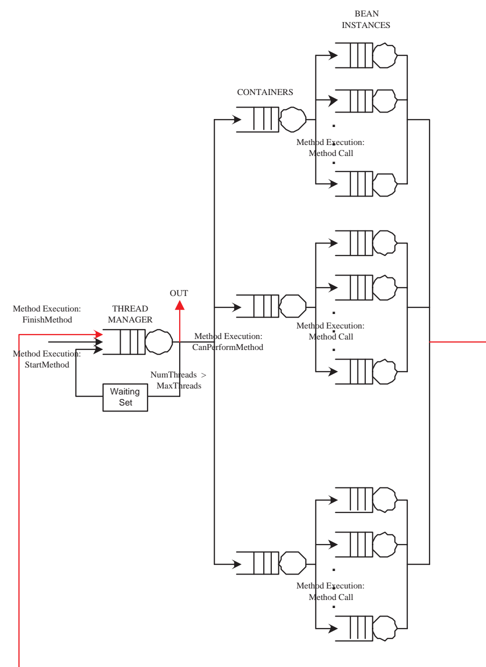

Figure 5: Global queueing network for a method execution

To simplify this system, we apply the Flow Equivalent Server method (FES) [5], which reduces the number of nodes by aggregating sub-networks into single, more complex (i.e. queue length dependent) nodes. Applying this method to our system, each FES sub-network consists of $M+1$ stations where 1 corresponds to the container for a bean class and M is as above. After short-circuiting, this sub-network results in the closed one shown in Fig. 6, which will be analysed to obtain its throughput.

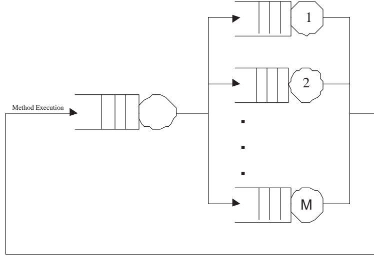

Figure 6: Short-circuited FES sub-network

The next step is to construct an analytical model corresponding to the FES sub-network in order to determine the service rate function for a FES node in the overall network. Blocking is a critical non-standard characteristic in the FES sub-network; a client who has completed service in the container station is blocked if the required bean instance is not active and there is no free instance to passivate – i.e. no idle server in the model. As a consequence, the blocking time needs to be calculated. In conventional blocking models, see for example [10], blocking time is normally equal to the residual service time at some downstream server. Here, however, it is the time required for the first of the M parallel servers to clear its queue in a blocking-after-service discipline.

4.1 The Aggregated Server and Whole Model

The complete (sub)model of a subsystem comprising a bean container and its M instance servers (i.e. bean method execution servers) at constant population N is shown in Fig. 7. The throughput function (as a function of N), once validated, will be used to parameterise the service rate of a FES for the subsystem in an aggregated model of the whole network shown in Fig. 5.

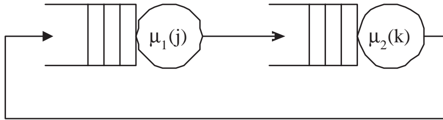

Figure 7: Complete Model

When there are j clients at the outer (container) server and $k = N-j$ at the M parallel servers, the service rate functions $\mu_1(j)$ (with blocking) and $\mu_2(k)$, are estimated as follows:

$$\mu_1(j) = \begin{cases} 1/(m_1 + \beta_{N-j} b(N-j)) & \text{if } (N-j) \geq M \\ 1/m_1 & \text{otherwise} \end{cases} \quad (1)$$

where m_1 is the mean service time for server 1 (the outer server) when there is no blocking, $b(N-j)$ is the mean blocking time when there are j customers at the outer server ($N-j$ customers at the parallel servers) and β_{N-j} is the dynamic blocking probability, which we derive in the next subsection. In [4], it is shown that $b(k) = k/(M^2\mu)$ by first deriving the blocking time probability distribution, under certain conditions that have been shown to be reasonable in this modelled system.

$$\mu_2(k) = \sum_{n=1}^{M} \xi_{kn} n\mu \quad (2)$$

where μ is the service rate of each of the parallel servers and I is the total number of instances available to each client. The parameter ξ_{kn} is the probability that there are n out of M busy servers, given that there are k customers at the parallel servers altogether; it is also derived in the next subsection.

Clearly the visitation rate is the same for both servers. The steady state queue length probability distribution for this network – $p(j)$ for the state with j tasks at server 1 and $N-j$ at server 2 – is approximated by using a simple, explicit Markov model in section 4.3. Throughput $T(N)$ at population N is then given by

$$T(N) = \sum_{j=1}^{N} p(j)\mu_1(j) \ . \quad (3)$$

4.2 Instance-Active and Client-Blocking Probabilities

Let z_k denote the equilibrium probability that, at an instant when a task completes service at the container server (with rate $1/m_1$) with k tasks at the M instance servers, at least one of the instance servers is idle. Let α denote the probability that the instance required by a task completing service at the container server is active, i.e. that the task can immediately join that instance's queue (whether or not empty) and so is not blocked. Then the dynamic blocking probability for a task completing service at the outer (container) server when there are k tasks at the parallel servers is:

$$\beta_k = \frac{(1-z_k)(1-\alpha)\frac{m1}{m1+\beta_k b(k)}}{z_k + (1-z_k)\frac{m1}{m1+\beta_k b(k)}}. \quad (4)$$

This quantity is the ratio of the equilibrium probability flux from unblocked states with k tasks at the parallel servers to a blocked state, divided by the total flux from unblocked states. We approximate α by M/I, assuming the next arrival to the parallel servers requires each of the I instances with equal probability.

The parameter z_k can be estimated by considering a simple two dimensional Markov chain (K_t, Ξ_t) where, at time t, K_t represents the number of tasks in total at the M parallel servers and Ξ_t the number of non-empty queues, i.e. busy servers, out of the M. However, we use an even simpler submodel to estimate z_k: an M-state Markov chain Ξ_t^k for each population size $k \geq M$ at the parallel servers, where each state corresponds to the number of busy queues in the system. Clearly this is an approximate decomposition since K_t and Ξ_t are not independent, but it does capture the essence of the blocking that is present. This results in a relatively low probability of empty queues (when $k \geq M$ and $I \geq M$) since it will often be the case that a task is blocked when a server with only one task completes a service period, resulting in an empty queue which is instantaneously occupied by the unblocked task; the empty state is therefore not seen.

For population size $k \geq M$ at the parallel servers, let the equilibrium probability that $\Xi \equiv \Xi_\infty^k = l$ $(l = 1...M)$ be denoted by $\pi_k(l)$.

The submodel then has balance equations:

$$\pi_k(l)(m_1)^{-1}\frac{I-l}{I} = \pi_k(l+1)(l+1)\mu\frac{l}{k-1} \tag{5}$$

for $1 \leq l \leq M-2$. The fraction on the left hand side represents the probability of a task not choosing an instance at one of the l busy servers. The fraction on the right hand side is the probability that any given non-empty queue has length exactly one, when there are $l+1$ busy servers with k tasks in total. We estimate it as the ratio of the number of arrangements of $k-l-1$ tasks in l queues, to the number of arrangements of $k-l-1$ tasks in $l+1$ queues; after assigning one task to each queue, there are only $k-l-1$ left. Since the number of arrangements of n tasks in m queues is $\frac{n+m-1)!}{n!(m-1)!}$, the required ratio is $\frac{l}{k-1}$.

The final balance equation, for $l = M-1$, is

$$\pi_k(M-1)(m_1)^{-1}\frac{I-M+1}{I} = \pi_k(M)v_k M\mu\frac{M-1}{k-1} \tag{6}$$

where v_k is the probability of observing, just before a departure instant from the parallel servers when there are k tasks there in total, a task at the front of the container queue in a non-blocked state – i.e. receiving service with mean time m_1 (when all of the parallel servers are busy). Hence we estimate v_k by the ratio of mean container service time to the sum of this time and the mean time blocked when all instance servers are busy, appealing to the law of large numbers for Markov chains. The time spent blocked is 0 if the required instance is active (with probability α) and $b(k)$ otherwise. Thus we define

$$v_k = \frac{m_1 M^2 \mu}{m_1 M^2 \mu + (1-\alpha)k} \tag{7}$$

The recursive function derived from the balance equations (5) can be written,

$$\pi_k(n) = \begin{cases} 1 & \text{if } n = 1 \\ \frac{(I-n+1)(k-1)}{n(n-1)m_1\mu I}\pi_k(n-1) & \text{if } 2 \leq N < M \\ \frac{(I-n+1)(k-1)(m_1 M^2\mu+(1-\alpha)k)}{M^2 m_1^2 \mu^2 I n(n-1)}\pi_k(n-1) & \text{if } n = M \end{cases} \tag{8}$$

Normalising the $\pi_k(n)$ to give the probabilities

$$\xi_{kn} = \frac{\pi_k(n)}{\sum_{l=1}^{M}\pi_k(l)} \tag{9}$$

, we now estimate z_k by $1 - \frac{\xi_{kM}v_k}{1-\xi_{kM}(1-v_k)}$ and β_k follows for $M \leq k < N$.

4.3 The Steady State Queue Length Probabilities

In order to represent the high variability in service times – tasks either have short service times (mean value m_1) if they are not blocked or very long ones otherwise – we consider the model shown in Fig. 8 where the outer server is split into two phases. Phase 1 comprises the normal service time at the container server (mean value m_1) and Phase 2 comprises the blocking time (with state dependent mean as derived in [4]). Consequently, when a customer finishes Phase 1 it goes to Phase 2 with probability β_k and goes to the aggregated node for the parallel servers with probability $1-\beta_k$, when there are k tasks at the parallel servers. Clearly only one of the phases is busy at a time.

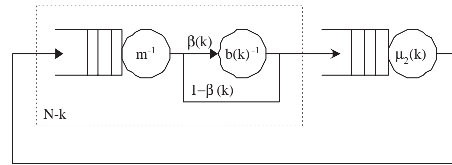

Figure 8: Model with a 2-phase outer server

The Markov chain (K_t, B_t) derived from this model can be seen in Fig. 9, where K_t is the number of tasks at the parallel servers and $B_t = 0$ (respectively, 1) if the task at the front of the container queue is blocked (respectively, not blocked), at time t. (For $K_t = N$, $B_t = 1$ with probability 1.) The parameter $b(k) = \frac{k}{M^2\mu}$ (see [4]), μ_2 is as in Eq. 2 and β_k is as defined in section 4.2. This chain is finite and irreducible and so has a steady state from which the equilibrium queue length probability p_j is computed as $P(K_\infty = N-j, B_\infty = 0) + P(K_\infty = N-j, B_\infty = 1)$.

5. NUMERICAL VALIDATION

Since it is only possible to analytically model complex distributed systems using several simplifications and assumptions, approximation techniques have to be – and have been – used. Consequently, theoretical results need to be validated by comparing them with those obtained from simulation. A simulation program was written using QNAP2 V. 9.3, a modelling language developed for INRIA (*Institut Nacional de Reserche en Informatique et Automatique*). Amongst other facilities, this language provides a discrete-event simulator. The simulations were run for 100,000 time units, using the spectral method, and the resulting confidence intervals had an error of less than 5% at a 95% level of confidence. In addition to simulation, validation against a 2-dimensional Markov chain (see Appendix A) that represents explicitly the behaviour of the container submodel has been carried out. Validation of the analytical model's predicted throughput and mean response time for the whole system is discussed below.

5.1 Throughput

Graphs of the throughput predicted by the analytical and simulation models for different values of N (number of threads), and M (number of paral.lel servers) are shown in Fig. 10 and Fig. 11. The parameters $m_1 = 0.4$ and $\mu = 1/4.1$. These graphs show excellent agreement between the analytical model and the simulation, the error being less than 3% at all points.

5.2 Queue Length Probability Distribution

Queue length probability distribution graphs are shown in Fig. 12 and Fig. 13. The simulation reveals a fairly sharp

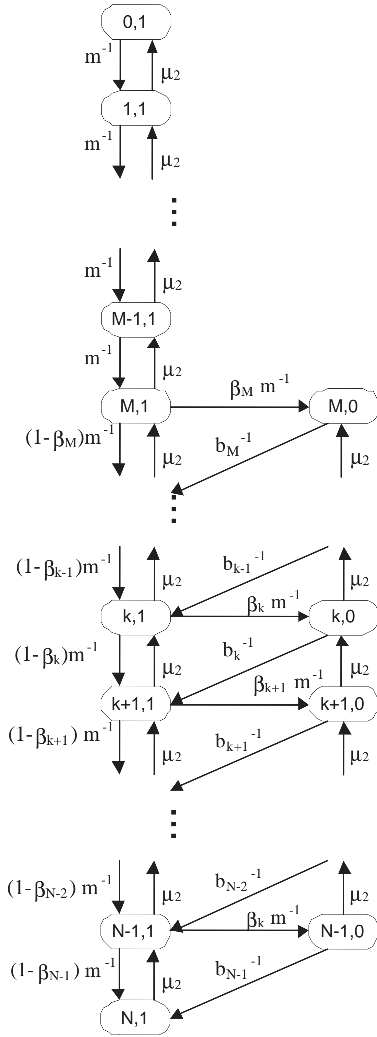

Figure 9: Markov Chain

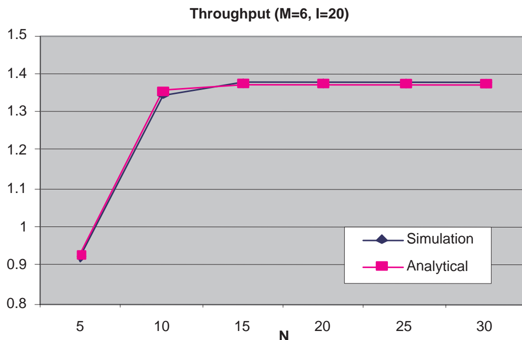

Figure 10: Throughput comparison for simulated and analytical results for $M = 6$, $I = 20$ and $N : 5-30$

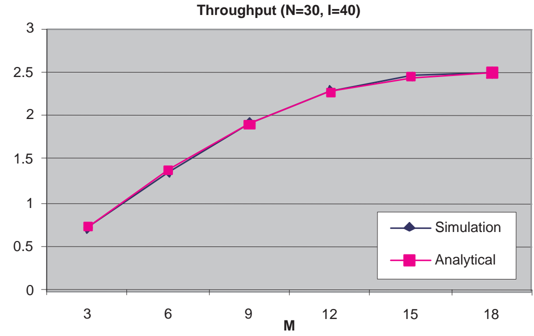

Figure 11: Throughput comparison for simulated and analytical results for $N = 30$, $I = 40$ and $N : 5-30$

peak at values of k just over M (j just below $N - M$). This is due to the onset of blocking. When $k < M$, there is no blocking and server 1 is very fast relative to server 2; hence the queue length probability is small but increasing (as server 2 gets faster). As soon as blocking occurs, increasingly as $k = M, M + 1, \ldots$, server 1 becomes the slower and so its queue length increases. This build-up is rapid since, when $k \geq M$, the probability of empty queues at the parallel server is reduced compared to a FES submodel in which all state vectors for the M queues have the same equilibrium probability. This is a consequence of the blocking mechanism; whenever all the instance servers are busy, there is a significant probability (viz. $1 - \alpha$) that the next arrival will be blocked and so immediately replace a departing task that leaves an empty queue.

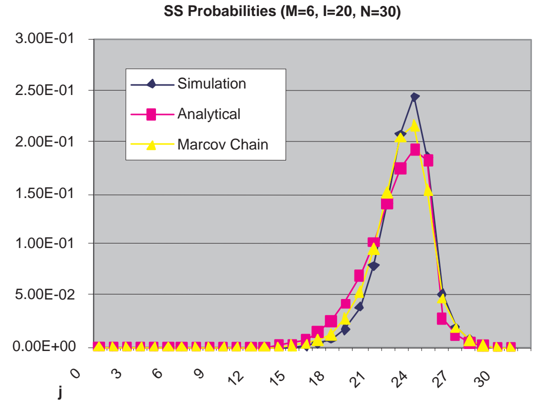

Figure 12: Steady state probabilities comparison for simulated, analytical and 2-D Markov chain results for $M = 6$, $N = 30$ and $I = 20$

The initial analytical model in [4] predicts this peak poorly, even though the blocking mechanism was represented in detail through estimates for α, z_k and β_k and hence $\mu_2(k)$. However, the variability of service times at the container server, when blocking is possible, was poorly modelled since the server was characterised solely by its rate, i.e. its *mean* service time. The present model captures this variability explicitly and produces dramatically better predictions.

As expected, the values obtained in the numerical solu-

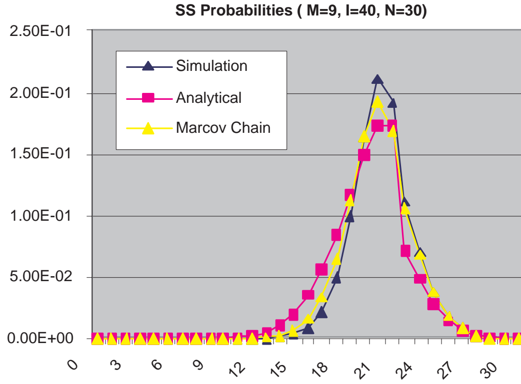

Figure 13: Steady state probabilities comparison for simulated, analytical and 2-D Markov chain results for $M = 9$, $N = 30$ and $I = 40$

tion of the 2-dimensional Markov chain stay in between the simulated values and the analytical values. Our analytical model uses the same assumption as the Markov chain model (same probability distribution for the different client-configurations in the non-idle parallel servers) as well as some other approximations. Consequently its predictions cannot be any better than the ones obtained through the Markov chain.

5.3 System performance results

The throughput functions of section 5.1 were used as the service rate functions of the FES in a reduced model of that shown in Fig. 5. Each container server and its associated M instance servers were replaced by a single FES node in standard fashion. We anticipate that this will be a good approximation, in contrast to the case of modelling method call executions at containers, since there is no blocking present. The system and its model are in fact analogous to those of multi-access systems successfully modelled by FES-decomposition techniques in the 1970s; see, for example, [5].

The prediction of the analytical and simulation models for the throughput and mean response time of the whole system were plotted in the graphs og Figs. 14, 15, 16 and 17 against system populations N in the range 5-40. The other model parameters were $C = 4$ (number of container servers, modelled by FES nodes), $I = 20$ (number of instances available per class),and $\nu = 16$ (service rate for the thread manager server). As can be seen from the graphs, results for two different types of configurations have been obtained. One configuration is symmetrical with $M = 6$ (number of instance servers per container) for all the containers (Figs. 14 and 15), and the other one is asymmetrical with $M = 2$ for one of the containers, $M = 4$ for two of them and $M = 8$ for the other one, giving the same total number of instance servers as the symmetrical case (Figs. 16 and 17).

The results for the symmetric case are unsurprising, since the visitation rates of the thread manager, containers and instance servers are respectively 24, 6 and 1. Hence the corresponding loads are 24/16 = 3/2, 6*(0.4 + mean blocking time) and 4.1. The thread manager is therefore not the

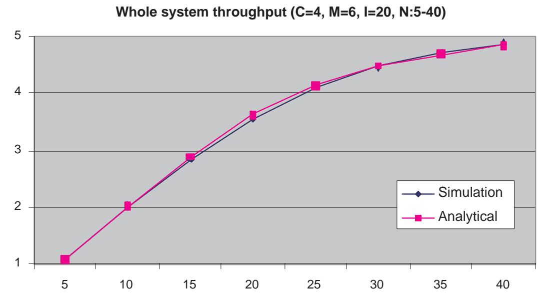

Figure 14: Symmetric case: Throughput comparison for simulated and analytical results for the whole system when $M = 6$, $I = 20$, $C = 4$ and $N : 5 - 40$

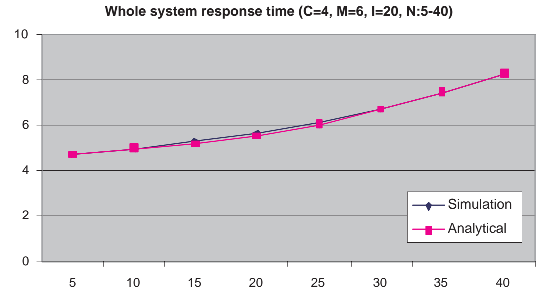

Figure 15: Symmetric case: Response time comparison for simulated and analytical results for the whole system when $C = 4$, $M = 6$, $I = 20$ and $N : 5{-}40$

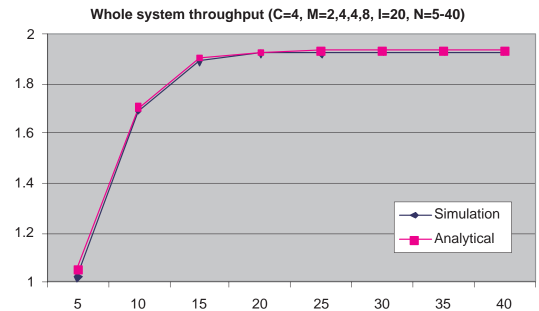

Figure 16: Asymmetric case: Throughput comparison for simulated and analytical results for the whole $C = 4$, $M = 2 - 4 - 4 - 8$, $I = 20$ and $N : 5 - 40$

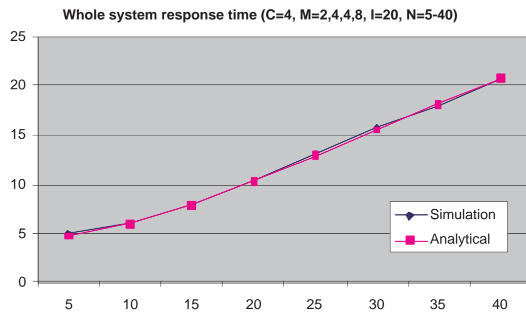

Figure 17: Asymmetric case: Response time comparison for simulated and analytical results for the whole system when $C = 4$ $M = 2-4-4-8$, $I = 20$ and $N : 5-40$

bottleneck and at relatively low populations (less than the total number of instance servers) there will be little contention and hence an almost linear performance improvement as population increases.

In the asymmetric case, the system overloads much more quickly as the population N increases. In this model, the instance servers at each container are equally utilised and each container is selected with the same probability 0.25. Therefore, the instance servers in the pool of two jointly form the bottlenecks which saturate at much lower population than in the symmetrical case.

6. CONCLUSION

We have developed an analytical, queueing-based model suitable for describing the performance of large-scale, distributed, three-tier, client-server systems supported by object-oriented technology; for example the EJB architecture. The approach we have followed is to study isolated functionalities of the system and then combine them in a hierarchical methodology. We started by analysing the behaviour of a method execution call and were able to isolate the blocking that occurs when an instance is inactive and all servers are busy, yielding an approximate, two-server, aggregated model.

Numerical results showed excellent agreement with simulation, especially at large M, where contention is less. However, even at high contention the error was less than 3%, suggesting a good building block for the modelling of complex, distributed, object-based scheduling systems.

Immediate future work will focus on certain improvements that can be made in the accuracy of some of our model's parameters. The Kensington Enterprise Data Mining System at Imperial College [2], [3] is a real example of a distributed client-server architecture of the type analysed. Using it, we intend to collect real performance data to validate our models with respect to an operational environment, and hence to use the models to guide the quantitative design of future generations of Kensington.

7. REFERENCES

[1] S. S. Alhir. *UML in a nutshell.* O'Reilly, 1998.

[2] J. Chattratichat, J. Darlington, Y. Guo, S. Hedvall, M. Kohler, and J. Syed. An architecture for Distributed Enterprise Data Mining. *High-Performance Computing and Networking*, 1999.

[3] I. C. D. M. Group. Kensington Enterprise Data Mining. http://Kensington.doc.ic.ac.uk.

[4] P. Harrison and C. M. Lladó. Performance Evaluation of a Distributed Enterprise Data Mining System. *Leture Notes in Computer Science: Computer Performance Evaluation: Modelling Tools and Techniques; 11th International Conference; Tools 2000.*, Mar. 2000.

[5] P. G. Harrison and N. M. Patel. *Performance Modelling of Communication Networks and Computer Architectures.* Addison-Wesley, 1993.

[6] W. Knottenbelt. *Parallel Performance Analysis of Large Marcov Models.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1999.

[7] S. Microsystems. Java Language Specification. http://java.sun.com/docs.

[8] S. Microsystems. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi.

[9] S. Microsystems. Enterprise JavaBeans 1.1 Architecture. http://java.sun.com/products/ejb/newspec.html, 1999.

[10] H. Perros and T. Altiok, editors. *Queuing Networks With Blocking.* North-Holland, 1989.

APPENDIX

A. EXPLICIT 2-D MARKOV CHAIN FOR THE CONTAINER SUBMODEL

The Markov chain with state transition graph shown in Fig. 18 represents explicitly the behaviour of the container submodel (see Fig. 6) and it has been solved numerically using Knottenbelt's Markov chain analyser [6]. Since the state space is finite and the chain is irreducible (as can be seen by inspection) a steady state solution for the state probabilities always exists. The solution of this Markov chain is very expensive when N and/or M are reasonably big and it is therefore only used for validation purposes.

Figure 18: 2-Dimensional explicit Markov chain