Performance Evaluation of a Distributed Enterprise Data Mining System

Peter G. Harrison and Catalina M. Lladó[⋆]

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queens Gate, London SW7 2BZ, United Kingdom
{cl16,pgh}@doc.ic.ac.uk

Abstract. We consider the performance of a distributed, three-tier, client-server architecture, typical for large, Java-supported, Internet applications. An analytical model is developed for the central schedulers in such systems, which can be applied at various levels in a hierarchical modelling approach. The system involves a form of blocking in which clients must wait for one of a number of parallel 'instance servers' to clear its outstanding work in order that a new instance may be activated. Thus, blocking time is the minimum of sojourn times at the parallel queues. We solve this model for the probability distribution of blocking time and obtain a simple formula for its mean value. We then use this result in a flow-equivalent server model of the whole system and compare our approximate results with simulation data. This numerical validation indicates good accuracy for the blocking approach *per se* as well as for system throughput, the performance objective chosen for the exercise.

1 Introduction

Distributed client-server systems, such as data mining systems, have become so complex that they cannot be designed efficiently without the use of some form of quantitative (performance) model; engineering experience alone is now too unreliable. Simulation can sometimes be used for this purpose but analytical models are preferred where possible in view of their greater efficiency and flexibility for experimentation. They make approximations in order to obtain tractable solutions and can be validated initially, in simple applications, by simulation models. In this paper, we develop an analytical model for a distributed, three-tier, client-server architecture, typical for large, Java-supported, Internet applications. In particular, we focus on the central schedulers in such systems, giving sub-models that can be applied at various levels in a hierarchical modelling approach.

The Kensington Enterprise Data Mining system is just such an architecture, whose application server is the critical performance component. The aim of the present research is the performance modelling and evaluation of this system and

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

specifically focuses on the behaviour of the application server. Data mining, or knowledge discovery in databases, is concerned with extracting useful new information from data, and provides the basis for leveraging investments in data assets. It combines the fields of databases and data warehousing with algorithms from machine learning and methods from statistics to gain insight into hidden structures within the data. Data mining systems for enterprises and large organisations have to overcome unique challenges. They need to combine access to diverse and distributed data sources with the large computational power required for many mining tasks. In large organisations, data from numerous sources need to be accessed and combined to provide comprehensive analyses, and work groups of analysts require access to the same data and results. For this purpose, the existing networking infrastructure, typically based on Internet technology, becomes a key issue.

The requirements for enterprise data mining are high-performance servers, the ability to distribute applications and the capacity to provide multiple access points [2]. To fulfil these requirements, Kensington Enterprise Data Mining embodies a three-tier client-server architecture which encapsulates the main application functions inside an application server that can be accessed from clients on the network. The three tiers are client, application server and database servers, as shown in Fig. 1. In addition, since enterprise data mining environments require flexibility and extensibility, the Kensington solution uses distributed component technology.

Under this paradigm, the system has been designed using the Enterprise JavaBeans (EJB) component architecture and has been implemented in Java. Databases anywhere on the Internet can be accessed via Java Database Connectivity (JDBC) connections and the application executes in the Kensington EJB server, an EJB architecture implementation.

The research carried out so far investigates one of the server functionalities which is the entity method execution call. The objective is to analytically model

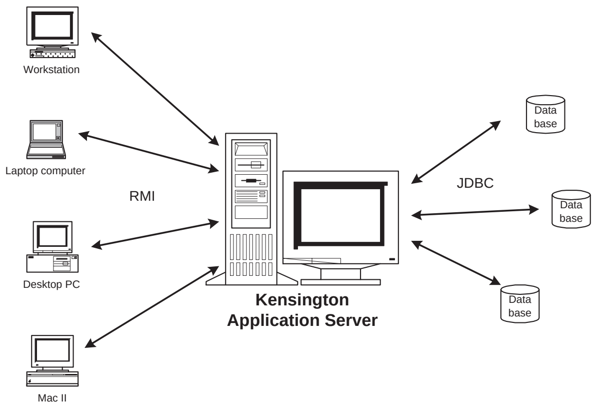

Fig. 1. Kensington Enterprise Data Mining Architecture

the method call behaviour in order to predict its performance, e.g. its throughput or mean response time. To study the behaviour of a distributed system, a model involving a number of service centres with jobs arriving and circulating among them according to some pattern is normally needed; typically a queueing network is appropriate. Therefore, the first stage of the investigation is to analyse the method call execution process in terms of queueing network concepts. Following this, the entity method execution call process is simulated and the results thus obtained used to validate the analytical model.

The rest of this paper is organised as follows. Section 2 describes the Kensington EJB server and its characteristics as an implementation of the EJB architecture. Section 3 explains the entity method call execution process and its analytical model is developed in Section 4. Numerical results are presented in Section 5 and the paper concludes in Section 6, also considering future directions for investigation.

2 Kensington EJB Server

The Kensington EJB server fully implements the EJB-1.0 specification [6] and also supports entity beans (defined in the draft EJB-1.1 standard). The EJB architecture is a new component architecture, created by Sun, for the development and deployment of object-oriented, distributed, enterprise-level applications.

2.1 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 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. 2 shows the class structure.

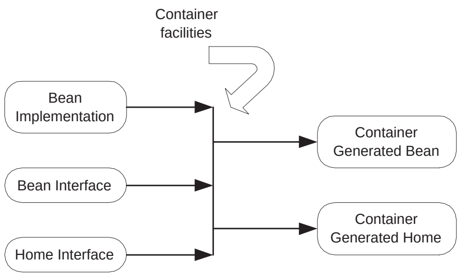

Fig. 2. Class generation using container facilities

2.2 EJB Server Implementation

In the Kensington EJB implementation, 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. 3 shows how a client uses different bean instances of the same bean class.

The communication between clients and EJB server is done through *Remote Method Invocations* (RMI), which is a protocol allowing Java objects to communicate [5]. A method dispatched by the RMI runtime system to a server may or may not execute in a 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 Kensington 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 in Section 2.2 below. Simi-

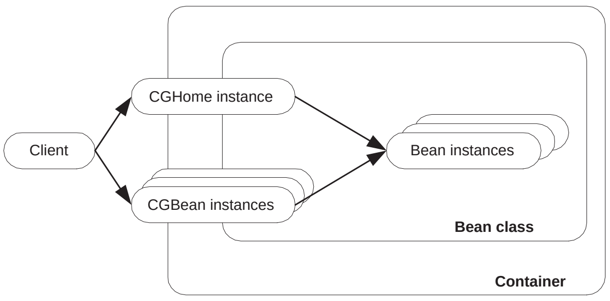

Fig. 3. Client use of several bean instances

larly, the characteristics and behaviour of entity beans are described in more detail in Section 2.2.

Java Synchronisation. In order to synchronise threads, Java uses monitors [4], 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.

Entity Bean Implementation. An entity container has a limited number of bean instances existing concurrently, i.e. 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. 4, 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.

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

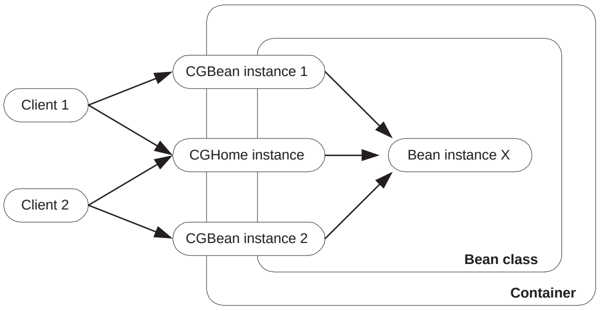

Fig. 4. Concurrent access from two clients

at the same time. In addition, CGBean instances use the thread manager which has some synchronised methods. 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. 5.

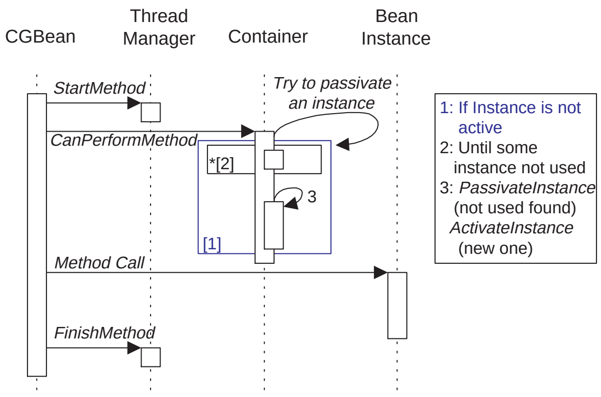
CGBean
Thread Manager
Container
Bean Instance
StartMethod
Try to passivate an instance
CanPerformMethod
*[2]
3
[1]
Method Call
FinishMethod
1: If Instance is not active
2: Until some instance not used
3: PassivateInstance (not used found) ActivateInstance (new one)

Fig. 5. Interaction diagram for a Method Execution Call

4 An Extended Queueing Model

Based on the behaviour explained above, a queueing network model is shown in Fig. 6. 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. 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. 6) 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, [3]), 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.

To simplify this system, we apply the Flow Equivalent Server method (FES) [3], 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. 7, which will be analysed to obtain its throughput.

The next step is to obtain the 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. As a consequence, the blocking time needs to be calulated. In conventional blocking models, see for example [7], 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.

The sub-model corresponding to the FES sub-network is appropriate to represent the behaviour of various functionalities of this system — and others — at various levels in the modelling hierarchy.

4.1 Blocking Time

Given that a customer is blocked at the container server, so that all the instance servers have non-empty queues, the blocking time B is the minimum of the times for each of the M queues to become empty, with no further arrivals during the time B. Let the random variable T_i denote the time for queue i to become empty, given that initially (i.e. when the customer is first blocked) there are $n_i \geq 1$ other customers in queue i, including the one in service ($1 \leq i \leq M$). Then the probability distribution function, $F_{n_i}(t)$, of T_i is Erlang-n_i with parameter μ_i, i.e.

$$F_{n_i}(t) = P_{n_i}(T_i \leq t) = 1 - \sum_{k=0}^{n_i-1} \frac{(\mu_i t)^k}{k!} e^{-\mu_i t} \qquad (n_i > 0)\ . \tag{1}$$

Now, the conditional (on queue lengths $\mathbf{n} = (n_1, \ldots, n_M)$) blocking time $T_{\mathbf{n}} = \min_{1 \leq i \leq M} T_i$ has complementary cumulative probability distribution function

$$P_{\mathbf{n}}(T_{\mathbf{n}} > t) = \prod_{i=1}^{M} P(T_i > t) = \prod_{i=1}^{M} \sum_{k=0}^{n_i-1} \frac{(\mu_i t)^k}{k!} e^{-\mu_i t} \tag{2}$$

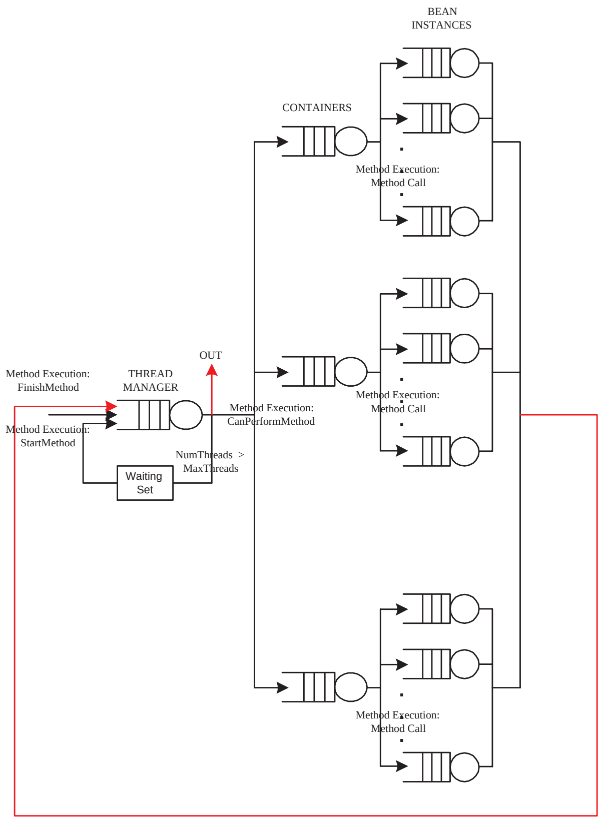
BEAN INSTANCES
CONTAINERS
Method Execution: Method Call
Method Execution: Method Call
Method Execution: Method Call
OUT
Method Execution: FinishMethod
THREAD MANAGER
Method Execution: StartMethod
Method Execution: CanPerformMethod
NumThreads > MaxThreads
Waiting Set

Fig. 6. Global queueing network for a method execution

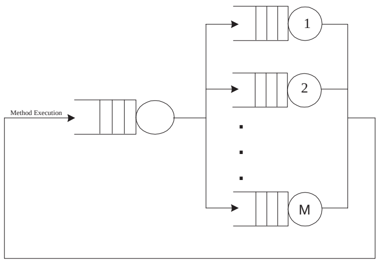

Fig. 7. Short-circuited FES sub-network

since the T_i are independent. Hence the unconditional blocking time has distribution function $B(t)$ given by

$$\overline{B}(t) \equiv 1 - B(t) = \sum_{\mathbf{n}:\forall i.n_i>0} \pi(\mathbf{n}) \prod_{i=1}^{M} \sum_{k=0}^{n_i-1} \frac{(\mu_i t)^k}{k!} e^{-\mu_i t} \tag{3}$$

where $\pi(\mathbf{n})$ is the equilibrium probability that there are n_i customers in queue i at the instant a customer becomes blocked at the container server. We make the approximating assumption that all valid states $\mathbf{n}$ (i.e. with $n_i > 0$ for all i) have the same equilibrium probability. This property would hold were we to have the job observer property in decomposable, Markovian network with equally utilised servers $1, \ldots, M$, e.g. in the case that there was no blocking and we were considering departure instants from the container server. With this assumption, when the population of the servers $1, \ldots, M$ is K, i.e. when $\sum n_i = K$, for all $\mathbf{n}$, $\pi(\mathbf{n})$ is the reciprocal of the number of ways of putting $K - M$ balls into M bags — at least one must be in each bag. In other words, for all $\mathbf{n}$,

$$\pi(\mathbf{n}) = \pi = \frac{(K-M)!(M-1)!}{(K-1)!} \; . \tag{4}$$

We now have the following result for the probability distribution of blocking time.

Theorem 1 *With the assumptions stated above, the probability distribution of the blocking delay suffered by a blocked customer at equilibrium is*

$$B(t) = 1 - \frac{(K-M)!}{(K-1)!} e^{-M\overline{\mu}t} \sum_{k=0}^{K-M} \frac{(K-k-1)!}{(K-M-k)!k!} (M\overline{\mu}t)^k \tag{5}$$

when there are K customers at the M parallel servers, where $\overline{\mu}$ is the average service rate of the M parallel servers, i.e. $M\overline{\mu} = \sum \mu_i$.

Proof When there are K customers in queues $1, \ldots, M$,

$$\overline{B}(t) = \sum_{\substack{\mathbf{n}: \\ \forall i.n_i>0,\ \sum n_i = K}} \pi \sum_{\substack{\mathbf{k}: \\ \forall i.0 \le k_i \le n_i - 1}} \prod_{i=1}^{M} \frac{(\mu_i t)^{k_i}}{k_i!} e^{-\mu_i t}$$

$$= \pi \sum_{\substack{\mathbf{k}: \\ \forall i. k_i \geq 0,\ \sum k_i \leq K-M}} \quad \sum_{\substack{\mathbf{n}: \\ \forall i. n_i \geq k_i+1,\ \sum n_i = K}} \prod_{i=1}^{M} \frac{(\mu_i t)^{k_i}}{k_i!} e^{-\mu_i t} \tag{6}$$

by changing the order of summation. The inner summand is independent of $\mathbf{n}$ and the number of terms is equal to the number of ways of putting $K - M - k$ balls into M bags, when $k = \sum_{i=1}^{M} k_i$. (The reasoning here is that $k_i + 1$ at least must be in each bag, using up $k + M$ of the K balls in total.)

Consequently, we may write:

$$\overline{B}(t) = \frac{(K-M)!(M-1)!}{(K-1)!} \times$$

$$\sum_{\substack{\mathbf{k}: \\ \forall i. k_i \geq 0,\ \sum k_i \leq K-M}} \frac{(K - \sum k_i - 1)!}{(M-1)!(K-M-\sum k_i)!} \prod_{i=1}^{M} \frac{(\mu_i t)^{k_i}}{k_i!} e^{-\mu_i t}$$

$$= \frac{(K-M)!}{(K-1)!} e^{-M\overline{\mu}t} \sum_{k=0}^{K-M} \frac{(K-k-1)!}{(K-M-k)!k!} \sum_{\substack{\mathbf{k}: \\ \forall i. k_i \geq 0,\ \sum k_i = k}} k! \prod_{i=1}^{M} \frac{(\mu_i t)^{k_i}}{k_i!}$$

$$= \frac{(K-M)!}{(K-1)!} e^{-M\overline{\mu}t} \sum_{k=0}^{K-M} \frac{(K-k-1)!}{(K-M-k)!k!} (M\overline{\mu}t)^k \tag{7}$$

and the result follows. ■

We assume that all the M servers are equally likely to be chosen by each arrival and that their service rates are all equal to μ — this guarantees equal utilisations. We then have

$$\overline{B}(t) = \frac{(K-M)!}{(K-1)!} e^{-M\mu t} \sum_{k=0}^{K-M} \frac{(K-k-1)!}{(K-M-k)!k!} (M\mu t)^k \ . \tag{8}$$

Mean blocking time, b say, which we use in the equilibrium model for the queue lengths for the whole system below, now follows straightforwardly.

Corollary 1 *Mean blocking time for blocked customers at equilibrium is $K/(M^2\overline{\mu})$ when there are K customers at the M parallel servers.*

Proof

$$b = \int_0^\infty \overline{B}(t)\mathrm{d}t = \frac{(K-M)!}{(K-1)!} \sum_{k=0}^{K-M} \frac{(K-k-1)!}{(K-M-k)!k!} \int_0^\infty (M\overline{\mu}t)^k e^{-M\overline{\mu}t}\mathrm{d}t \ . \tag{9}$$

Changing the integration variable from t to $s/M\overline{\mu}$ then yields

$$b = \frac{(K-M)!}{M\overline{\mu}(K-1)!} \sum_{k=0}^{K-M} \frac{(K-k-1)!}{(K-M-k)!k!} \int_0^\infty s^k e^{-s}\mathrm{d}s$$

$$
\begin{aligned}
&= \frac{(K-M)!}{M\overline{\mu}(K-1)!}\sum_{k=0}^{K-M}\frac{(K-k-1)!}{(K-M-k)!} \\
&= \frac{(K-M)!}{M\overline{\mu}(K-1)!}\sum_{k=0}^{K-M}\frac{(k+M-1)!}{k!} \qquad (10)
\end{aligned}
$$

(by changing the summation variable to $K-M-k$.)

We complete the proof by showing by induction on N that

$$\sum_{k=0}^{N}\frac{(k+m)!}{k!} = \frac{(N+m+1)!}{(m+1)N!} \qquad (11)$$

for $N, m \geq 0$. For $N=0$, both sides are $m!$. Now assume that

$$\sum_{k=0}^{n}\frac{(k+m)!}{k!} = \frac{(n+m+1)!}{(m+1)n!} \qquad (12)$$

for $n \geq 0$. Then we have

$$
\begin{aligned}
\sum_{k=0}^{n+1}\frac{(k+m)!}{k!} &= \frac{(n+m+1)!}{(m+1)n!} + \frac{(n+m+1)!}{(n+1)!} \\
&= \frac{(n+m+1)!}{(m+1)(n+1)!}(n+m+2) \qquad (13)
\end{aligned}
$$

as required. Substituting $N = K-M$ and $m = M-1$ now yields

$$b = \frac{(K-M)!}{M\overline{\mu}(K-1)!}\frac{K!}{M(K-M)!} = \frac{K}{M^2\overline{\mu}} \qquad (14)$$

which completes the proof. ■

4.2 The FES and Whole Model

The complete (sub)model of a bean container and its M bean method execution servers at constant population N is shown in Fig. 8 and detailed below.

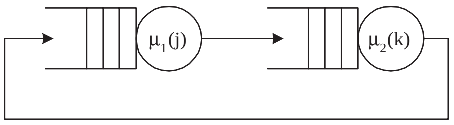

Fig. 8. Complete Model

The service rate functions $\mu_1(j)$ (with blocking) and $\mu_2(k)$, where $k = N-j$ (for the FES) are defined as follows:

$$\mu_1(j) = \begin{cases} \left(m_1 + \beta_{N-j}\frac{N-j}{M^2\mu}\right)^{-1} & \text{if } (N-j) \geq M \\ {m_1}^{-1} & \text{otherwise} \end{cases} \qquad (15)$$

where j is the number of clients in the outer (container) server and m_1 is mean service time for server 1 (the outer server) when there is no blocking. The parameter β_{N-j} is the blocking probability, which we derive in the next subsection.

$$\mu_2(k) = \begin{cases} \frac{Ik}{I+k-1}\mu & \text{if } I < M \\ z_k(M-1)\mu + (1-z_k)M\mu & \text{if } I \geq M,\ k \geq M \\ \frac{Mk}{M+k-1}\mu & \text{otherwise} \end{cases} \tag{16}$$

where k is the number of clients at the M parallel servers and μ is the service rate of each of them. The parameter z_k is the probability that there is at least one idle server; it is also derived in the next subsection.

Clearly the visitation rate is the same for both servers. The steady state probability distribution for this network – $p(j)$ for the state with j tasks at server 1 and $N-j$ at server 2 – is then calculated as a product form in standard fashion (see, for example, [3]). Throughput T is then given by

$$T = \sum_{j=1}^{N} p(j)\mu_1(j)\ . \tag{17}$$

4.3 Instance-Active and Client-Blocking Probabilities

Let α denote the probability that the instance required by a task arriving at the M parallel servers is active, i.e. that the task can immediately join that instance's queue (whether or not empty) and so is not blocked. We approximate α by M/I. Let z_k denote the equilibrium probability that, when there are k tasks at the M servers altogether, at least one of them is idle. Then the blocking probability is $\beta_k = (1-z_k)(1-\alpha)$.

The parameter z_k can be estimated by considering a simple two dimensional Markov chain (K_t, Z_t) where, at time t, K_t represents the number of tasks in total at the M parallel servers and $Z_t = 0$ if there is at least one empty queue amongst the parallel servers, $Z_t = 1$ if not. A more precise model would replace Z_t by E_t, the number of empty queues, i.e. idle servers, out of the M. However, we use an even simpler submodel to estimate z_k: a two state Markov chain Z_t for each population size k at the parallel servers. Clearly this is an approximate decomposition since K_t and Z_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 $k \geq M$) 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 never seen.

For population size k at the parallel servers, let the equilibrium probability that $Z \equiv Z_\infty = 1$ (respectively $Z = 0$) be denoted by $\pi_k(1)$ (respectively $\pi_k(0)$). Obviously, for $k < M$, there is always an empty queue and so $\pi_k(0) = 1$ and $\pi_k(1) = 0$. For $N > k \geq M$, the balance equations for the submodel are then

$$\pi_k(0)m_1^{-1}u\frac{I-M+1}{I} = \pi_k(1)m\mu\frac{m_1M^2\mu}{m_1M^2\mu + k(1-\alpha)} \tag{18}$$

where, at population k, u is the probability that there is exactly one empty queue, given that $Z = 0$, m is the average number of singleton queues (i.e. with exactly one task) given that $Z = 1$ (no empty queues). The fraction on the left hand side represents the probability of a task not choosing an instance at one of the $M - 1$ busy servers. The fraction on the right hand side is the probability of observing a task at the front of the container queue in a blocked state, as opposed to receiving service with mean time m_1.

Now, by symmetry, u is the probability that queue number 1 is the *only* empty queue, given that it is empty, and so may be estimated as the ratio of the number of arrangements of k tasks in $M - 1$ queues with at least 1 task in each, to the number of arrangements of k tasks in $M - 1$ queues. This can be written,

$$u = \frac{k!(k-1)!}{(k-M+1)!(k+M-2)!} \ . \tag{19}$$

Notice that if $M = 2$, $u = 1$ and if $M = 3$, $u = (k - 1)/(k + 1)$ as required.

Next, $m = Mv$ where v is the probability that queue number 1 (for example) has length exactly one, given that its length is at least one. Thus we estimate v as the ratio of the number of arrangements of $k - M$ tasks in $M - 1$ queues, to the number of arrangements of $k - M$ tasks in M queues; after assigning one task to each queue, there are only $k - M$ left. Hence, $v = \frac{M-1}{k-1}$ and $m = \frac{M(M-1)}{k-1}$ for $k \geq M$.

We now estimate z_k by $\pi_k(0)$ and β_k follows for $M \leq k < N$.

5 Results

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*). Among other facilities, this language provides a discrete-event simulator.

The simulations were run for 100,000 time units, using the batch means method, and the resulting confidence intervals had an error of less than 5% at a 95% level of confidence. The graphical representation of the simulation and analytical results for different values of I (number of bean instances), N (number of threads), and M (number of paral.lel servers) is shown in Fig. 9, Fig. 10 and Fig. 11 respectively. The parameter r relating to sequences of calls to the same instance was set to 0.8, $m_1 = 0.4$ and $\mu = 1/4.1$.

6 Conclusion

The Kensington Enterprise Data Mining system is a real example of a distributed client-server system. As such, its complexity makes its analytical modelling non-trivial. The approach we have followed is to study isolated functionalities of the

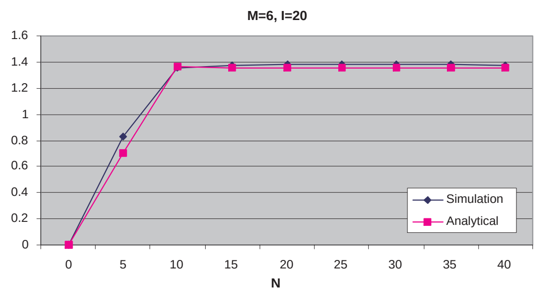

Fig. 9. Throughput comparison for simulated and analytical results for M=6, I=20 and N:0-40

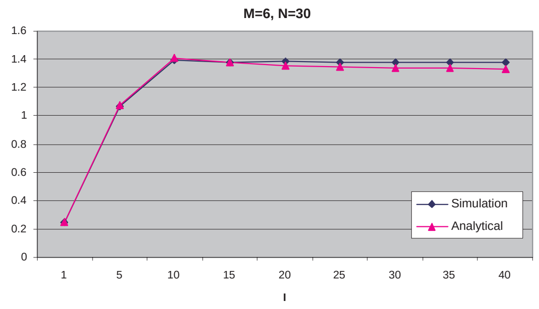

Fig. 10. Throughput comparison for simulated and analytical results for M=6, N=30 and I:1-40

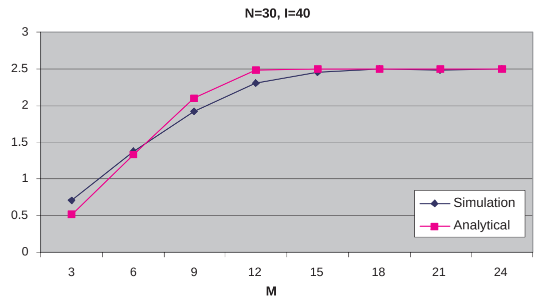

Fig. 11. Throughput comparison for simulated and analytical results for N=30, I=40 and M:3-24

system and then combine them in a hierarchical methodology. We started by analysing the behaviour of a method execution call and divided the network into two sub-networks (FES and complement sub-networks). In this way, we were able to isolate the blocking that occurs when an instance is inactive and all servers are busy, yielding an approximate two-server model.

Numerical results showed generally good agreement with simulation, especially at large M and I, where contention is less. However, even at high contention the error was less than 5%, suggesting a good building block for the modelling of complex, distributed, object-based scheduling systems.

Future work will focus on studying other functionalities of the system, where some of the results and techniques used in our method execution analysis will be re-applied. In particular, we will focus our attention on a more detailed model of the outer layers of Fig. 6 in which a more conventional form of blocking after service exists. There are also certain improvements that can be made in the accuracy of some of our model's parameters. In particular, a more detailed Markov model is being developed to determine z_k and hence the service rate function of the FES.

References

1. Sinan Si 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. Peter G. Harrison and Naresh M. Patel. *Performance Modelling of Communication Networks and Computer Architectures.* Addison-Wesley, 1993.
4. Sun Microsystems. Java Language Specification. http://java.sun.com/docs.
5. Sun Microsystems. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi.
6. Sun Microsystems. Enterprise JavaBeans 1.0 Architecture. http://java.sun.com/-products/ejb, 1998.
7. H.G. Perros and T. Altiok, editors. *Queuing Networks With Blocking.* North-Holland, 1989.