QoS Control of E-business Systems through Performance Modelling and Estimation

Thesis by

Giovanna Ferrari

A thesis submitted for the degree of

Doctor of Philosophy(PhD)

at Newcastle University

Newcastle University

School of Computing Science

Newcastle upon Tyne (UK)

July 2007

To Cesare.

Acknowledgements

It is a pleasure to thank the people who have provided guidance and support during my time at the School of Computing Science at Newcastle. Colleagues and friends here have warmly welcomed me and made me forget how rainy the summer can be in Newcastle.

In particular, I would like to express my sincere gratitude to my PhD supervisor Dr. Paul Ezhilchelvan, who has supported and encouraged me with sound advices, solid knowledge, constructive ideas, good company and lots of patience throughout the thesis work.

I would also like to thank Professor Santosh Shrivastava, for believing in me and taking me on-board the TAPAS Project, and Professor Isi Mitrani, for his valuable lessons and wise suggestions.

A heartfelt thank goes to Antonio, Doug, Jennie and Qurat-ul-Ain, who have been excellent colleagues, but, above all, precious friends. It has been great fun working with you.

Finally, a special thank to Cesare, who has cooked the most delicious dinners while I was busy writing this thesis up.

Abstract

E-business systems provide the infrastructure whereby parties interact electronically via business transactions. At peak loads, these systems are susceptible to large volumes of transactions and concurrent users and yet they are expected to maintain adequate performance levels. Over provisioning is an expensive solution. A good alternative is the adaptation of the system, managing and controlling its resources.

We address these concerns by presenting a model that allows fast evaluation of performance metrics in terms of measurable or controllable parameters. The model can be used in order to (a) predict the performance of a system under given or assumed loading conditions and (b) to choose the optimal configuration set-up for certain controllable parameters with respect to specified performance measures.

Firstly, we analyze the characteristics of E-business systems. This analysis leads to the analytical model, which is sufficiently general to capture the behaviour of a large class of commonly encountered architectures. We propose an approximate solution which is numerically efficient and fast. By mean of simulation, we prove that its accuracy is acceptable over a wide range of system configurations and different load levels.

We further evaluate the approximate solution by comparing it to a real-life E-business system. A J2EE application of non-trivial size and complexity is deployed on a 2-tier system composed of the JBoss application server and a database server. We implement an infrastructure fully integrated on the application server, capable of monitoring the E-business system and controlling its configuration parameters.

Finally, we use this infrastructure to quantify both the static parameters of the model and the observed performance. The latter are then compared with the metrics predicted by the model, showing that the approximate solution is almost exact in predicting performance and that it assesses the optimal system configuration very accurately.

Contents

List of Figures

Chapter 1

Introduction

The pervasiveness of the Internet offers a convenient platform for E-business activities, such as on-line banking, auctioning and retailing. In this scenario, an E-business system offers services to its customer who demands — and pays — for both the service and a certain *quality* of the provided service. Hence, an important aspect of the service relationship is the set of Quality of Service (QoS) guarantees that the provider must meet in order to satisfy customers.

Meeting QoS guarantees is a challenging task, mainly due to the unpredictability of the E-business system and variations of the average load. One solution to this problem would be to size the system for the peak load, but this could be too expensive [41]. A good alternative is to allow the adaptation of the system, managing and controlling system resources according to the change of conditions. After all, as Charles Darwin said, *"It is not the strongest of the species that survives, nor the most intelligent, but rather the one most responsive to change"*. The adaptation of a system makes it more suitable for existence under the conditions of its environment.

A first step toward system adaptation is to provide an analytical model that allows evaluation of performance metrics in terms of measurable or controllable parameters. This is exactly what we present in this thesis. Through analysis of the model, we obtain a usable and efficient approximate solution, the accuracy of which is evaluated by comparing the model estimates with those obtained from simulations. Furthermore, we calibrate the model and assess the correspondence of the model estimates with those observed on a real system. We use our implementation of a QoS Control System to quantify both the static parameters of the model and the observed performance, and to aid the tuning of the configuration set-up during the testing phase. We examine the effect of several controllable parameters on the performance, showing that the approximate solution is almost exact in predicting the performance of the E-business system and that it assesses the optimum system configuration very accurately.

In the remainder of this chapter we describe the background issues that motivated our work. Then we articulate the thesis statement and indicate the contributions introduced by this thesis. Finally, we outline the content of each chapter.

1.1 Background and Motivation

This thesis deals with the issue of estimating and controlling QoS of E-business systems. The thesis stems from the work done in the EU-funded TAPAS project [45]. Some of the objectives of this project were: (a) to elaborate notations for expressing Service Level Agreements (SLAs) to enable specification of QoS and (b) to develop QoS-enabled middleware services capable of meeting SLAs of hosting applications. Middleware services are those services implemented to support development and delivery of complex distributed applications.

QoS requirements are stipulated within SLAs established between the service provider and its customers. The SLA is a legally binding contract, which defines both the services to be provided and the metrics that determine the successful delivery of these services [44].

QoS metrics can be analyzed from different viewpoints. For instance, a user's perception of good performance has to do with fast response time and no connection refused; on the other hand, the management's perception of performance also includes high throughput. Metrics that are commonly considered to be important for an E-business system [41] are: *average response time* (the time taken to process an accepted request), *throughput* (the average number of requests processed per second) and *rejection probability* (the long-term fraction of requests that are rejected because the load is too heavy).

An E-business system provides some services to its customers, using middleware technologies to deploy, host and manage applications for enabling marketing, ordering and payment transactions. Such a system can be susceptible to large volumes of transactions and concurrent users and yet it is expected to maintain adequate performance levels. The economic impacts of bad performance in E-business are not to be overlooked. Poor QoS, such as a high response time during peak hours, translates into frustrated customers and leads to lost business opportunities.

Regarding middleware architectures, the software industry currently focuses on component based technologies, such as J2EE. These technologies provide an integrated environment for the application execution, offering ready to use E-business services such as transactions, persistence and security. While on the one hand these technologies simplify the development of E-business applications, on the other hand they introduce high levels of complexity in estimating the performance. The multilayered structure of the architecture, the background services and the tied coupling of application components with the component infrastructure are all factors that complicate performance analysis.

Finally, E-business systems are provided with several tunable parameters that also influence performance. The system administrator manually tunes these parameters to optimize the performance, in many cases using a mixture of trial-and-error and best practice approaches. However, there are cases[49] in which the best configuration of the server is found to be in conflict with the best practice suggestion.

Hence, the main questions arising are:

- Is it possible to build a model that accurately predicts the performance and, at the same time, remains applicable to a wide variety of of E-business systems?
- There is a wide set of variables that influence the performance of the platform itself. Will it be possible to clearly understand which of these adjust and how, for the purpose of achieving the established QoS?

A solution to these problems may have important practical implications, since it would offer a good basis for managing and configuring the E-business system at any time.

1.2 Thesis Statement and Objectives

We believe it is important to provide a performance model which describes the way system resources are used in processing client requests.

The model has to capture the main factors determining the behaviour of an E-business system in order to estimate and optimize performance, even in the presence of load variations. Using this model, a set of equations can be defined, which returns the QoS metrics of interest in terms of measurable or controllable parameters. This will in turn give support to system administrators for dynamically selecting an appropriate parameter setup that would match the application level load and, at the same time, attain the desired QoS level.

The objectives of this thesis include:

- To assess the size and capacity of hardware and software resources needed for providing E-business services, establishing operating baselines and trends.
- To predict system performance, determining average and maximum QoS metrics for a given workload and a given platform.
- To identify potential bottlenecks and tune the configuration in order to support the achievement of the best performance of the overall system at any time, without

leaving resources under-utilized.

1.3 Contributions

This thesis introduces the following set of contributions:

1. A model that allows a accurate evaluation of QoS metrics in terms of measurable or controllable parameters and support for system management. The model captures the behaviour of a large class of E-business systems, since it is not tied to a specific technology or application. Besides, it is sufficiently detailed to take into account the essential system features.

2. An efficient approximate solution that is accurate in representing system operations but also simple enough to be numerically tractable. The price paid for the generality of the model is that it cannot be solved exactly, thus we provide the approximate solution, whose accuracy is evaluated by comparisons with simulations.

3. The evaluation of the approximate solution against a real-life E-business system, constituted by a J2EE application (the *ECperf* benchmarking application), the JBoss application server and a database server. Firstly the model is calibrated, then its predictions are compared with the performance observed on the real system. The results demonstrate that our model effectively conforms to reality.

4. A working solution for a QoS Control System, fully integrated in the popular JBoss application server platform. The system can be used to monitor the application server, to tune the configuration setup and to accomplish the testing phase for the model calibration process.

The novelty of the work resides in its inclusion of (a) a model that contains a new set of features compared with those taken into account by other studies, (b) an efficient analytical solution validated by means of both simulation and calibration, which estimates the effect of several controllable parameters on the performance of a real system and (c) an infrastructure that enhances with QoS control functionalities a popular open-source application server.

1.4 Thesis Outline

The thesis is structured as follows:

Chapter 2 identifies the wider body of literature to which this thesis contributes, exploring the relevant research made in the area of performance control for E-business systems.

Chapter 3 analyses the activities within an E-business system by representing these in an abstract yet realistic manner, and also by postulating a set of rules which govern the interactions between them. The analysis leads to a model that captures the behaviour of a large class of E-business applications and technologies.

Chapter 4 presents the analytical model and its approximate solution. This solution provides a trade-off between the accuracy in representing E-business system operations and computational tractability.

Chapter 5 evaluates the accuracy of the approximate solution by means of a series of numerical and simulation experiments. The graphs presented here compare the behaviour of the simulated system with the approximate solution over a range of system configurations.

Chapter 6 describes the implementation details of our QoS Control System which monitors and modifies the configuration parameters of an application server. It is explained how JBoss extensibility characteristics are exploited to integrate the QoS Control System in the application server platform.

Chapter 7 illustrates the calibration process, during which several sets of experiments are carried out with the use of the QoS Control System. The experiments validate the approximate solution against a real system, showing a close match between estimations and observations.

Chapter 8 concludes the thesis summarizing the achievements. In addition the chapter provides an overview of future works and possible extensions.

Chapter 2

Related Work

2.1 Introduction

In this chapter we illustrate how the scientific community has tackled the problems of bounding QoS metrics and controlling the performance of E-business systems under different conditions.

We review related work in this area, beginning with presenting a taxonomy of the main approaches as illustrated in Figure 2.1.

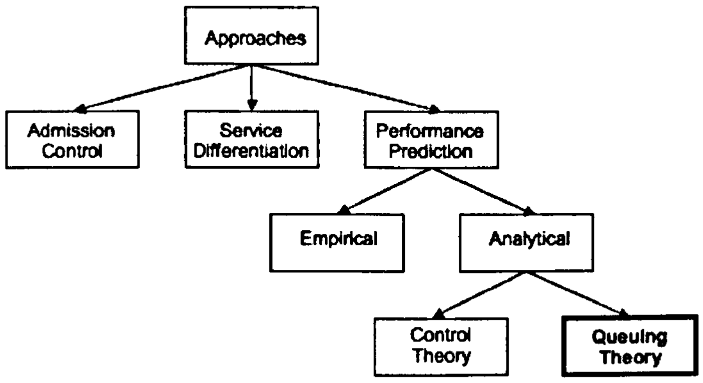


Figure 2.1: Taxonomy of the Approaches

These approaches may be succinctly described as follows. We first describe the *Admission Control* and *Service Differentiation* techniques. The former simply prevents systems from being overloaded discarding part of the requests; the latter classifies customers and deliver the best service to one customer class at the expenses of the others. Then we focus our attention on the *Performance Prediction* approach, which relies on analysis of the E-business platform behaviour. The objective is to predict performance and adapt or reconfigure the system toward a desired QoS. The analysis can be *Empirical*, i.e. based exclusively on test experiences, or *Analytical*, i.e. based on models which evaluation could be supported by tests. Specifically, the two main methods used for modelling are *Control Theory* and *Queuing Theory.*

In our work we apply the Queuing Theory within the Analytical Approach, the last section of this chapter justifies the choice and indicates our contributions.

2.2 Admission Control

Admission Control is employed to prevent systems from being overloaded. As systems approach saturation, response time grows very large and throughput degrades substantially. To avoid this, a fraction of requests is explicitly discarded, rather than forcing all clients to experience unacceptable response times. Typically the choice of which request to discard is arbitrarily made.

Some of the works done in this area implement a "proxy-server", which transparently intercepts the client requests to be rejected, like the *Gatekeeper* proxy in [12] and *Yaksha* in [24].

The approach used by *Elnikety et al.*[12] requires a setup phase so as to determine the capacity of the system, which is then fed back into the admission control mechanism. Using a set of off-line tests, the authors determine the capacity of the system and measure the load generated by different request types, maintaining estimates of their expected service times. At runtime, a request is admitted to the system as long as it does not exceed the system capacity, otherwise the request is inserted in an admission queue, waiting until the system is not overloaded.

Kamra et al.[24] have a similar approach, the main difference lies in the use of a self-tuning proportional integral controller, whose design is based on the model abstraction of the E-business system. The controller automatically adapts to variation in load, rejecting the requests that exceed the load, therefore there is no need of an off-line measurements phase. Both approaches don't require any modification of the platform, but are completely transparent to the E-business system and the running application.

Welsh and Culler[64] apply a range of techniques in the *SEDA Web Server* to make maximal effort to avoid rejection. The SEDA framework decomposes system services into multiple stages, where each stage is a component performing part of the request process divided in events. At each stage a monitor measures the response time and a controller decides to admit a given event to that particular service. If the event is not admitted, alternative actions can be executed, hence the user's request is not systematically rejected, but, for instance, part of it is redirected to another node. The fine-grained admission control mechanism enables to monitor single resources and to reject only those events leading to bottlenecks.

In [8]*Cherkasova et al.* the granularity of rejection is not the request, but whole "sessions" (sequences of individual requests made by the same client and needed to complete a transaction). The authors observe that overloaded web server discriminates against longer sessions, while this approach guarantees a fair probability of completion for any accepted session, independent of its length. The predictive admission control strategy accepts a new client session only if the server has the capacity to successfully complete all the related requests. Instead, if the server is close to being overloaded, the entire session is rejected.. Estimation of the server capacity and evaluation of the load generated by the sessions is computed using a basic queuing network simulation model.

2.3 Service Differentiation

Service Differentiation can be considered as a special case of Admission Control, discriminating one set of customers against others. The objective here is to distinguish between different classes of customers, such as premium, business and economy classes, so that the high priority class receives a preferred service. In this way limited resources can be effectively used, even under overload conditions.

This approach is applied by *Bhatti et al.*[5], who implement the server architecture *WebQoS*. Incoming requests are classified and reordered based on scheduling policies, then submitted to the server for normal processing. For instance, requests from premium users are denoted high priority and given preferential processing treatment, hence are executed as high priority processes by the host operating system. Different service-level policies can then be applied to each category. For example, if the server is experiencing heavy traffic, low priority requests can be redirected to an alternative server or receive rejection notices.

Kang et al.[25] present a real-time database QoS management framework for E-business applications. The framework provides performance guarantees in terms of differentiated deadline miss ratio and data freshness. User requests are classified into classes according to their importance, each class receives different guarantees on its miss ratio to support real-time data services. When overload surges, the system performance is improved by providing preferred services to the high priority classes.

Another example is the work by *Lu et al.*[37], who achieve differentiated delay guarantees by dynamic connection scheduling. Significant connection delay arises when all server processes are tied up with existing connections, in which case new connection requests wait in the TCP listen queue to be accepted by a server process. A connection scheduler component allocates server processes to waiting connections according to priority of the class. Based on a theoretical model, the scheduler controls that "process budgets" of all classes are not overwhelmed.

A similar approach is taken by *Abdelzaher et al.*[1]. They also define a theoretical model of web server to estimate per-class response time and throughput guarantees in the presence of load unpredictability. The Service Differentiation technique here is associated with Content Adaptation, so that the quality of the delivered content is adapted to the load conditions, for instance, in case of overload, the quality of a retrieved file is degraded omitting the multimedia content.

2.4 Performance Prediction

Performance prediction is the process of estimating performance measures of a computer system for a given set of parameters. The computer platform running the E-business system is analyzed in order to predict its behaviour at different load conditions or at the variation of the system parameters.

The system parameters are characteristics of the computer platform or intrinsic features of a resource. Determining the relevant parameters and setting them appropriately can have a significant effect on the performance. The parameter setting can be tuned up (a) at run-time (*dynamically*), e.g. the configuration of multiple thread pools, queues, timeouts; or (b) at server down-time (*statically*), e.g. by increasing the CPU speed rating, switching Java Virtual Machine (JVM) or modifying cache options. The system configuration can be adjusted to optimally adapt at the variation of the conditions.

So far two approaches have been followed to predict and optimize E-business system performance: the *Empirical Approach* and the *Analytical Approach.* The former approach examines the system behaviour by means of a series of tests. The latter uses tests to support or validate the correctness of an analytical model, which describes the system based on a set of analytical equations. With these equations performance metrics are obtained from model parameters, the analytical foundation is provided by mathematical techniques such as: *Control Theory* or *Queuing Theory.*

The following subsections illustrate the works that apply these approaches.

2.4.1 Empirical Approach

In the Empirical Approach, analysis are led by a series of tests to examine the behaviour of the system, identify the main bottlenecks and fine-tune the system. Tests are run with the support of specific applications purpose-built to simulate real applications, or using benchmarking applications, i.e. commercial tools that can emulate thousands of virtual clients operating as real users interacting with the system.

In general, to estimate E-business system performance is not straightforward. This is because the tight coupling of application components and component infrastructure introduces a high level of complexity in predicting the effects of various architectural trade-offs. The works mentioned here point out that the same technology performs differently from one instance to another if applied to (a) different vendors [6], (b) different classes of E-business applications [49], (c) same application but implemented using different methods [7], or (d) different configuration set-up [53, 51].

The technical report [6] presents a detailed analysis, evaluation and comparison of six leading application server technologies using a benchmark application. The report reveals that current component oriented architectures used for application servers are not equal. There are in fact significant differences in functionality, performance and scalability across all the products evaluated. Some of the differences become even more pronounced as the same application is scaled to run on more application server nodes within a cluster.

The works [49] and [65] address the problem of finding optimal system configurations for given applications. In [49] the authors use a simple methodology to explore the configuration space for two different classes of application, a manufacturing and a trade application, revealing different characteristics and requirements between them. For instance, the manufacturer application requires more database connections to perform better, whereas the trade application can do with fewer connections; also, the latter respond well if fewer application server threads are concurrently serving client requests and hence competing for resources.

This methodology is extended in [65]. The problem this time is formulated as a black box optimization problem. This is also known as "global optimization problem", where the goal is to obtain the global optimal solution, which, in the specific case, is the best configuration setting. What the authors are exploring here is an efficient search algorithm that obtains a good configuration setup with a minimum numbers of experiments. The one they propose is the Smart Hill Climbing algorithm, shown to be efficient in both searching and random sampling among several configurations.

Both these works highlight significant correlations among the parameters to tune and, at the same time, present best effort algorithms that can be used to increase the efficiency in selecting the configuration parameters.

Checchet et al.[7] investigate the behaviour of the middleware platform running a specific class of application, the RUBiS auction application [50], with five different J2EE implementation methods (a detailed description of the J2EE technology will be defined in Chapter 6). For the implementation using stateless Session Beans, communication cost forms the major component of the execution time on the EJB server and the design of the container has little effect on performance. With Entity Beans implementations the design of the container becomes important, in particular, the cost of reflection affects performance. For implementations using Session Façade Beans, local communication cost is critically important. The last implementation uses EJB with Local Interfaces, which improve the performance by avoiding the communication layers for local communications. They notice that, among those implementation methods, the method with Servlets running SQL generally scales better than the one using Session Beans with SQL, which in turn scales far better than the implementation with Entity Beans and Container Management Persistence (CMP).

On the contrary, the authors of [51] state that the conclusions of [7] are misleading due to some omitted configuration issues. They show that all implementation methods can perform equally well, provided that the configuration parameters are tuned properly. Moreover, the performance is dramatically influenced by the tuning of some configuration parameters, except that it is rather complex to predict the effects of composing the various tweaks and optimizations.

We have also provided our contribution to this debate, in order to offer some guidelines for developers. In [53] we consider the RUBiS application with five different client-side and execution environments. The resulting study identifies the strengths and the weaknesses of each implementation method under several scenarios. The scenarios we explore range from various database access methods to different cache configuration, from variable client workload to mutable think time. Indeed we observe that certain methods are more suited than others in given scenarios. For instance, the EJB with Local Interfaces implementation achieve the highest throughput in most of the cases. However, its performance is beaten by Stateless Session Beans and Entity Beans with Session Façade in the scenario with variable client think time. CMP Entity Beans seem to have the highest response time and it is the least recommended method in most cases. Java Servlets achieve their best performance in experiment focused on caching, but show a low throughput when it comes to database accesses and updates.

In [31] *Kounev et al.* study how the performance of JBoss application server can be improved by exploiting different configuration options offered by the JBoss platform. Firstly, they compare the three alternative web containers, but this revealed no significant performance differences. Significant performance gains are brought instead when Local Interfaces is used to access business logic components from the presentation tier. Also different caching configuration set-up introduce performance improvement. It is worth noting that those configuration options are specific to JBoss only. Finally, the authors demonstrated that switching to a different JVM may improve JBoss performance by up to 60%, unfortunately they do not disclose the details of the JVMs employed.

The performance data, gathered at runtime or during the experiments, is also used to extrapolate statistics which describe the system behaviour. This case is illustrated in [23], where the authors use machine learning techniques to determine if the system meets the required performance and to locate system bottlenecks. However, the empirical process employed is rather complex. At detection process, several tests are required in order to determine when exactly the system does not meet the SLA. The results of this process are used in the subsequent analysis for bottleneck detection. This is composed of three steps and only the last one employs machine learning to form the decision tree. This eventually reveals the potential bottleneck. During these stages there are approximately 200 metrics collected by the sensors, thus the metrics need a further analysis in order to reduce the amount of extraneous non-correlated metrics. Another example of this approach is described in [2]. In this case samples of load, throughput and response times are collected at runtime and analyzed in order to determine the relationship between variables representing the state of the system and the performance metrics. This process leads to a performance model, named "Historical Performance Model", which can be used to predict the behaviour of an E-business system on heterogeneous servers. This approach is then compared to one of the methods of the analytical approach, as it will be discussed in the following sections.

2.4.2 Control Theory

This Analytical Approach employs differential equations as fundamental modelling tool for the design of a *control system*, where, generally speaking, the objective is to make some output behave in a desired way by manipulating some input [11]. A control system is composed by a *controlled system* in combination with a *control loop*; interactions between them consist of observations and manipulations performed by the control loop on the controlled system. The control loop compares samples of performance achieved by the system against a specification of the expected performance and decides some actions to be taken in order to adhere to the expected behaviour. The main components of the control loop are: (a) the *sensor*, which collects samples from the controlled variables, (b) the *controller*, which measures the difference between the actual and the expected value of the variables and decides the control strategy to apply, and (c) the *actuator*, which carries out the control strategy, accordingly changing some variables.

Parek and al. [48], describe a methodology to design control systems for applying linear control techniques to software systems, a more difficult domain compared to mechanical systems. The sensor measures the length of client waiting queue in a Lotus Notes e-mail server, the controller takes as input the control error, i.e. the difference between the reference value and the measured value of queue length, to decide whether to adapt. Depending on the current and past values of the error, the *ServerMaxUser* parameter is adjusted, and the new value is computed with a complex integral equation.

The one described is an example of the common approach used in applying Control Theory to computing systems: it is centered on the Single-Input Single-Output (SISO) technique, in which a single control and a single variable are regulated. Another example is [1], described in Section 2.3, where the only controlled variable is the server utilization. In this case, the sensor monitors current server resource utilization, really using an estimation in order to smooth real measurements, which are very noisy in practice. The controller calculates server load, based on a liquid flow model, and determines a subset of clients that can be admitted without overloading the server. The actuator translates controller output into physical actions, applying Admission Control or Service Differentiation.

In some cases, where the system is Multiple-Input Multiple-Output (MIMO), the Control System is decomposed into multiple simpler and independent SISO control loops, like in [37], where controlled variables are the server threads response times of all classes. The controller computes process proportions for each class and instructs the adaptive connection scheduler to associate server processes with connections of different classes.

In many cases complex systems cannot be decomposed because of tight interactions between the controls. *Diao et al.*[10] deal with this problem and show an example in which a more general MIMO technique is used to enforce policies in the presence of such interactions. In order to meet CPU and memory utilization targets, the controller tunes two of the Apache server parameters, *MaxClient* and *KeepAliveTimeout*, the number of server processes and the per-connection idle timeout respectively.

It is worth pointing out that, while Control Theory is widely used to control a vast variety of physical systems, it is arduous to apply to standard software systems, due to the complexity on deriving linear model of the system behaviour. Linear models or approximations can not accurately describe systems liable to high variations of workload and resource requirements, and detailed information are impossible to keep, since at every software release the system may change completely. Besides, controller actions easily induce oscillations in system response, and thus increase variability.

2.4.3 Queuing Theory

The well-known Analytical Approach of Queuing Theory has been applied, among others, to performance modeling of web architectures. The system is represented by a Queuing

Network, a collection of service stations where jobs, such as client requests, are serviced. Service stations represent system resources and are formed by one or more servers that process requests.To define the model of a Queuing Network, some assumptions are needed, such as the nature of the arrival stream, time taken to process the request, routing of requests among service stations and scheduling strategies. The outcome is a model that can be solved analytically or numerically to obtain the different performance metrics[42].

The following studies differ from each other by the system features they choose to model and the simplifying assumptions they introduce in order to make the model tractable.

Kounev et al. [29] demonstrate that models built by Queuing Theory are powerful as a performance prediction tool for E-business systems, in particular they show how can be exploited for performance analysis. The application considered is the SPECjAppServer2001 [52] (successor of ECperf [59]), commonly used for measuring the performance and scalability of J2EE application servers. The model is a closed queing network model, i.e. for each instance of the model the number of concurrent clients sending requests to the system is fixed, with five classes of requests, one for each type of transaction generated by the application. The authors use the model in order to compute the number of application server nodes needed in the cluster to guarantee adequate performance and to identify the component representing a potential bottleneck (identified as the one with the highest utilization value).

In [61] a n-tier system is modelled by a closed queue of networks, where every tier is represented by a single state dependent server. A request can make multiple visits to each tier during its overall execution, each visit is a transition from a tier to its neighbour tier. This model also captures cache effects, since a cache hit causes the request to immediately return to the previous tier, with transition probability 1. Thus the impact of cache hits and misses can be incorporated by appropriately determining the transition probabilities and the service time. The model includes active sessions, modeling these with an infinite server queuing system, in which each server is occupied by an active session. These servers feed the network of queues and form the closed queuing system. A mean value analysis algorithm of the model computes the average response time of a request, given parameters such as average service time at the tiers, visit ratio, number of concurrent session and think time of a session. This model is general enough to capture the behaviour of E-business systems with an arbitrary number of tiers, however, it can not capture the impact of increased context switching overhead or contention for memory or locks, since these fine-grain features of the system are not modelled.

A similar model is used by *Menasce et al.*[38], where a closed queuing network model is used to define a three-tier E-business system. The set of equations to estimate the performance metrics is defined by formulating the queuing network model that describes a single tier, considering contention for both physical (CPU or disks) and software resources (threads or database locks) and queues that arise at each resource, as it further explained in [41]. Since the three tiers are described in the same manner, the overall queuing network model is made of a composition of three single tiers, along with the functional dependencies between the tiers. The model is used to study the system performance and to select the best combination of configuration parameters so that the desired performance level can be achieved, even in the presence of workload variations. A controller component is implemented, which monitors the E-business system and decides how to modify the configuration, represented by a vector of configuration parameters that can be dynamically changed in a range of values. Search on the space of configuration parameters is done with a greedy algorithm, the Hill Climbing. The controller computes the performance metrics for each configuration and eventually selects the configuration with the optimal performance.

Liu et al.[36] model an application server with finite thread pool capacity as a closed queuing network model. The model characterizes operations of service components and frequency of each operation, from which to determine the equation of the performance profile parameters. To calibrate the model, a generic application is deployed on the target platform and the parameter values are measured, describing the platform performance profile. Hence the parameters are substituted into the analytical model to calculate the required quantitative prediction of performance of that system. The extension of the single server model to a model for a two-node server cluster is described by two identical server machines, where the workload is balanced between them and they access a shared database server. The same model is used in [35] to determine the optimal concurrency level of an EJB application server. The server container processes the incoming request and dispatches it to the thread queue. The threads serving the queue initialize the bean instances for the requests and pass them through the container invocation chain, which synchronizes access to the database with the beans. Based on empirical results, they observed that there is an optimal concurrency level of the system, determined by the size of both the server thread pool and database connection pool, and it is a balance between the concurrency and the contention overhead. Another factor influencing contention level are the number of active database connections and the ratio of read-only and read-write transactions.

The difference between the models in [36] and in [38] is that, while the latter is more generalized, the former focuses on the software infrastructure of an application, thus a specific implementation method of an application must be considered to derive the equation parameters. The advantage of the approach in [36] consists on the fine granularity in modeling the request life, considering service time variables such as find, write, read, update. On the other hand, it requires a very detailed knowledge of the application to deploy, in terms of type of implemented transactions and number of Session, Entity or Message Beans. Moreover, [38] proposes a solution for a control problem, monitoring the input in order to control output variables, even in the case of environment variations. On the contrary, [36] presents an accurate model of the system, but omits further investigation on how to design a controller that adapts the system behaviour to the fluctuation of the workload.

Another formalism that can be used in the Analytical Approach is the Layered Queuing Network (LQN). This is a specialization of the conventional Queuing Networks, adapted to the description of traditional software architectures, such as client-server systems. The approximate solution of the model can be generated automatically using a LQN solver. An example of a LQN model is defined in [34] for predicting the performance of an EJB-based application framework. The EJBs of the application are modelled by three layers of "tasks". A task represents a server process and has one or more entries corresponding to the major method calls exposed to the clients. The EJB tasks have entries corresponding to the business methods in the first level, to the database management system in the second level, and to a disk subsystem in the third level. This structure is then used to describe the behaviour of the system when clients invokes the different methods on the EJBs. As for [36], this approach also requires a very detailed knowledge of the deployed application.

In [3] *Bacigalupo et al.* employ LQN to predict the response time of an E-business system based on the Websphere platform. A single FIFO waiting queue represents each application server, the database queue has a FIFO queue per application server, and both types of server can concurrently process multiple requests. Requests are sent by clients organized in service classes, each class represents a different method called by the client on a definite application. A workload manager routes the incoming requests to the application servers. The model is defined with application server, database server and database server disk layers, where each layer contains a queue and a processor. Model parameters, such as service class processing time and maximum concurrency at each server, are found by calibrating the model offline. The specific application used is the Trade benchmarking application[15]. Moreover, the predictive accuracy of the LQN model is compared to the historical model described in [2], showing that the two methods can be equally used to make predictions for E-business systems with a good level of accuracy.

Overall, Queuing Theory seems to be a powerful method to model system performance, the only drawback is that the model must meet certain conditions, which, if not well stated, can make the model deviate significantly from the real system, or can prevent it from capturing fine-grain features of the system.

Only a few works consider queuing networks not suitable for representing passive resources, like [30], which introduce an additional modeling formalism that integrates queuing networks with Petri Nets (QNP). The authors conduct a brief capacity planning study, characterizing the workload of the SPECjAppServer2001[52] application and analyzing a single case study, the "order entry" operation, composed of four different transaction types. Yet it is significant that the presented model, rather than describing the application or the platform on the whole, describes a specific operation, therefore it can not be used for a general approach. However, the same authors recognize the inherent limitations of QNP modelling, due to state space explosion problem, since are applicable only to systems of small size and complexity, whereas with Queuing Theory even complex models can be easily solved, since a large number of efficient methods is available for their solutions.

2.5 Summary and Contributions

The important problem of controlling and managing the performance of E-business systems has been addressed by a vast literature. Many approaches have been adopted and in this chapter we have provided a taxonomy that includes examples of the most popular.

In particular, in our thesis we choose to predict the performance of a system in relation to its structure and the external workload, rather than applying a policy that reduces the workload. Moreover, we decide to adapt the system at the variation of some condition, conveniently tuning the setting of some system parameters. This is similar to what [49, 65] have done. However, we differ from those works since we do not apply the Empirical Approach, which, in our opinion, is not entirely effective. This approach does not attempt to analyzes the system internal structure and it is limited by the fact that only situations similar to those encountered during the experiments can be effectively managed. Instead the approach is enriched when coupled with the Analytical Approach.

Therefore we address the Queuing Network method within the Analytical Approach, once the model is defined, then it is validated by experiments. Specifically, we consider Queuing Network since it is powerful in modelling system performance. It easily solves even complex models, having the use of a large number of efficient methods for their solutions.

The contribution of our work resides in a queuing network model which incorporates a new set of features compared with those taken into account by other studies. This features will be explained in details in Chapter 4, there were the analytical model is presented.

Compared to [61, 38], our model presents a finer granularity in describing the system internal structure and interactions, which is important for accurately describing the reality. At the same time, the model is sufficiently general to capture the behaviour of a large class of E-business applications and technologies, while [36, 3, 30] describe a specific implementation method or a definite application. Abstracting from implementation details is advantageous, since the technology progress quite fast and these details rapidly become obsolete. For instance, in the latest Java Bean technology, EJB 3.0, one of the two persistence mechanisms (the BMP) does not exist anymore, therefore the works studying in details this mechanisms could be of little use when the new technology will completely substitute the old one.

The analytical solution derived from the model is validated by comparison with both simulations and with a real world E-business system. In either cases we examine the effect of several controllable parameters on the performance, showing that the approximate solution is almost exact in predicting the performance of the E-business system and that assesses the optimum system configuration very accurately.

Finally, we propose a working solution for a QoS Control System that enhances with QoS control functionalities a popular open-source application server, JBoss. This system can be used to monitor the application server, to tune the configuration setup and to accomplish the testing phase for the model calibration process. To the best of our knowledge this is the only implementation of QoS Control System for the JBoss application server that can be used to estimate the performance and to predict the optimal configuration setup.

Chapter 3

Analysis of the System Model

3.1 Introduction

This chapter analyses the activities within an E-business system by representing them in an abstract yet realistic manner. The behaviour of a large class of E-business systems is examined and a set of rules governing the interaction within components is derived.

The objective is to be able to derive an analytical model that is *accurate* in representing system operations and also analytically *tractable*. The model will allow evaluation of Quality of Service (QoS) metrics in terms of measurable and controllable parameters.

The analytical model can then be used for controlling the QoS offered by systems of commonly encountered architectures. This will lead to the system administrators to dynamically selecting appropriate configuration parameters that would match the user's SLAs.

The chapter is organized as follows. Firstly we define the structure of a generic E-business system, generally organized in "tiers" with different uses, in which each tier can be maintained by one or more server nodes. The operations and the interactions between the tiers are illustrated by means of a common example of an E-business application. The typical request pattern captures the functionalities of each tier and denotes the focus of our model, which is centered on the application server tier. This tier is designed to be flexible and it is provided with "knobs" used to vary the configuration parameters, but to choose the correct settings is not straightforward. We conclude the chapter by illustrating the characteristics of some of those parameters which can be controlled in order to control the system performance.

3.2 E-business System Architecture

An E-business system is conventionally structured into logical components called *tiers*. A tier comprises one or more servers capable of running a particular set of applications. As shown in Figure 3.1, usually we can expect to have the following:

Presentation Tier. The first tier is the front end of the E-business system. The web server runs at this tier, maintaining the presentation logic of the application, for instance in the form of *static* HTML pages or servlets. The web server receives the client requests coming from the outside network and passes them on to be processed at the lower tier.

Business Tier. The middle tier is responsible for business specific computations. The application server within it maintains the enterprise objects and the associated business logic. For this reason the business tier deals with more complex and *dynamic* queries.

Data Tier. The last tier is represented by database servers which maintain the persistence data. This tier is usually physically separated by the other two in order to protect sensitive data by being accessed by unauthorized users. The users cannot directly connect to this tier, but must connect to the data through one of the front-end tiers.

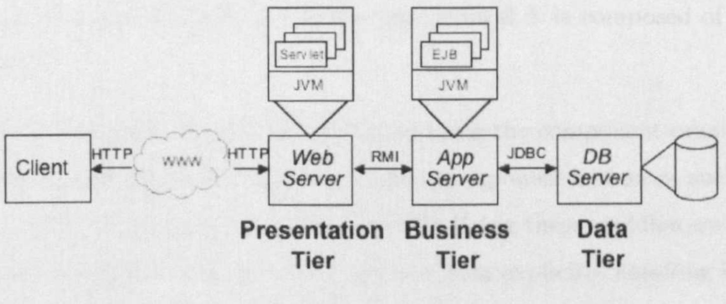


Figure 3.1: Structure of an E-business System

Note that some authors, such as [28], additionally define a *Client Tier* containing all potential clients accessing the system using the Internet. Very little computation is done at this tier, other than a client either composing a request or "thinking" over a response received and possibly submitting a subsequent one after some think time. We however focus here on those tiers that make up the system.

3.2.1 Tier Implementation Alternatives

For reasons of scalability, responsiveness and availability, a tier within an E-business system can be implemented by a *cluster* of server nodes. A *cluster* is a logical group of nodes running Web applications simultaneously and appearing as a single server to the world [60]. *Load balancers* are used to distribute the load between the cluster nodes by redirecting the traffic to an appropriate cluster member.

Tier-Cluster mapping. E-business systems can differ in the way the tiers are mapped to the clusters that implement them. In *single-tier clustering*, all three tiers are implemented by servers of a single cluster. A load balancer, placed in front of the Presentation Tier, distributes requests, hence the processing load, evenly to all servers in the cluster. A *multi-tier clustering* extends the logical separation between tiers into physical separation: every tier runs on a distinct cluster, with a load balancer at each tier. If the tier functionality is *replicable on-demand*, the number of servers within a cluster, and thus the provisioned tier capacity, can be varied dynamically.

Database Replication. With regard to the Data Tier, traditionally database servers have employed a *shared-nothing* architecture that does not support replication. However, some databases use the *shared-everything* approach, but the complexity of its distributed architecture is shielded by providing a transparent view of a single database image to external applications, even if it is composed of two or more servers[66].

Technologies. E-business systems are developed using the component-oriented technologies that promote the use of *containers* to host component instances, such as CORBA, COM+/.NET, J2EE and the recent Java EE. Using these middleware technologies, developers of E-business applications are free from explicitly handling issues such as transactions, database interactions and concurrency, which are handled instead by the Business Tier, the one providing the business logic. Selection of the architecture or the technology for E-business system implementation largely depends upon usage pattern and the type of the application.

3.3 Tier Operations and Interactions

A client on the Internet sends a request to the E-business system using, for example, a browser and thus connecting to the system web pages.

The request is always routed to the Presentation Tier, where a web server instance, or a thread, serves the request and generates either a static or a dynamic content response. In the latter case, the Business Tier is contacted to execute the program that contains the necessary business logic. The interaction is in the form of a blocking call being made to an application server thread, thus the web server thread awaits until the thread of the subsequent tier returns a response.

The application server thread, on its behalf, may have all the data necessary for the computation in the local cache. In that case it swiftly returns its response to the web server. If this is not the case, the data are retrieved from the database issuing one or more queries. Here again the application server thread is blocked while awaiting the database thread to serve the request. As a call returns, the caller may do further processing; at the end the Presentation Tier synthesizes the response that is returned to the client.

In the simplest case each request is processed exactly once by the tier and possibly it is directly returned to the caller; otherwise it is forwarded to the tier below for further processing. More complex processing at the tiers is also possible, in such scenarios, each request can visit a tier multiple times.

These interactions between the tiers are illustrated in Figure 3.2.

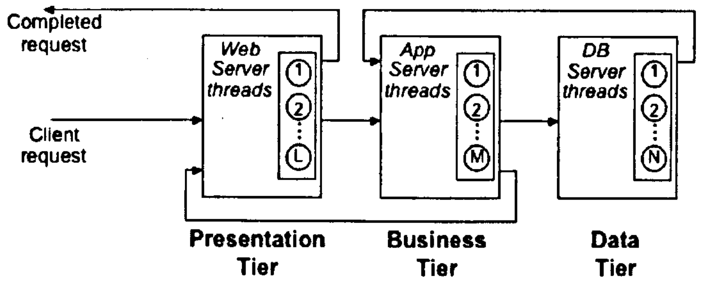


Figure 3.2: Tier Interactions in the E-business System

3.3.1 Example: a J2EE Application

To capture the functionalities of the tiers, we focus on a commonly used middleware architecture, the Java 2 Platform Enterprise Edition (J2EE)[55] and we consider the case of a specific E-business application, an auction.

An auction application implements the functionalities of an auction system, such as selling items, browsing and bidding. Selling and bid operations are usually restricted to the registered users whose credit data have been stored on the system database.

In a typical request pattern for a bid operation, the client sends a HTTP request to place a bid on a given item. A Java Server Page (JSP) or a servlet at the Presentation Tier processes the request; this triggers a series of invocations at the Enterprise Java Beans (EJBs) [54] deployed in the Business Tier. EJBs are the components that implement the application business logic. They are executed in the EJB containers, which provide a set of ready-to-use services including security, object persistence and transactions, more technical details will be presented in Chapter 6.

In order to display information on an item, the EJBs retrieve the data from the database, if they are not already cached; before registering the newly placed bid, it may also check the registration status of the bidder and her credit requirements. The EJB contacts the database using the JDBC service[56] of the container, which provides a connection pooling mechanism to facilitate connection reuse across requests. The number of connections in the pool is limited, if all connections are busy the requests waits in a queue. As soon as a connection becomes available, the data are retrieved from the database and returned to the EJB, which elaborates the response.

There are operations that require more than one access to the database. For instance, placing the winning bid entails checking the client's credit details, to finalize the purchase, to close the auction notifying the other bidders. All these are SQL operations at the database, to search, read or write data on the database tables. Some of the accesses, such as the final purchase, can be performed using a *transactional database access*: the database checks for conflicts as part of acquiring a write lock, and, if there are conflicts, i.e. read or write locks, then it will not get the lock.

In case lock conflicts occur on the database, the transaction being attempted is automatically rolled back and an exception will be throwing to the application. Another possible implementation is to block the thread when a conflict is detected, until the current locks are released, due to commit or roll back of the other transactions. The behaviour after a transaction failure depends on the type of implementation: in the *Container Managed Persistence* (CMP) the transaction is controlled by the application server, which immediately rolls backs the transaction; on the other hand, in the *Bean Managed Persistence* (BMP) the bean controls the transaction, handling the returned exception, thus the effect of a failure can be different than an error response, e.g. a retry.

There are two types of transactions possible. *Local transactions* are those that run purely in the scope of the database and commit or rollback within a single database interaction. *Global transactions* are those that are controlled within the application server and may encompass other application servers and databases. A local transaction that is used within a global transaction is automatically subordinate to the global transaction, i.e. it cannot commit or roll back independently.

The final computation of the EJB is transmitted to the calling JSP page that synthesizes the HTML response sent to the client and even contacts the other bidders to declare the winner of the auction.

The given example points out the specifics of the three tiers:

- *Specifics of the Presentation Tier:* the only task of the Presentation Tier is to handle HTTP requests, contact the business components and eventually generate HTML pages. In [63], the authors stated that the amount of work performed by the web servers is low enough that over-provisioning is a cheap solution to meet QoS requirements.

- *Specifics of the Business Tier:* this tier implements the business logic using the container components that manage the creation and execution of the EJBs and handle issues such as EJBs lifecycle, security, transactions, database interactions, concurrency control, messaging, naming and management support. The application server controls the behaviour of the EJBs, their concurrency and their caching, as well as the interactions of the EJBs with the data sources. The application server is connected to a back-end database that maintains persistent data. The data required by the computation are fetched from the database, when not cached locally. If the computation modifies the data, then the updates are implemented at the database by means of a transaction. If there are more calls to the application server than the available threads, the excess calls are queued.

- *Specifics of the Data Tier:* the database system maintains the persistent data and it executes the SQL queries as it is required by the EJBs. The complexity of this tier is usually hidden from the E-business application, infact the issues such as security, database open connections and database transactions are services managed by the container components.

It is worth noting that the type of workload for E-business traffic is basically different from the general web traffic. The main differences lie in: (a) presence of high level of online transaction processing activity, (b) large proportion of dynamic requests and (c) great number of requests in secure mode. For these reasons higher response time is triggered at the Business tier, caused by increased computation and communication times combined with the time spent to access the database.

The application server reduces the burden on the database by not requiring the database to directly manage session information and it allows more complicated session data than is practical to pass with every web request. It provides communication with both back-end database and front-ent web clients in addition to providing a framework to connect application specific "business rule" that governs the interaction between the two[26].

It is possible to improve the databases ability to meet its imposed QoS requirements. Since there are methods to maximize profits in the Presentation and in the Data Tiers, an E-business system profit depends on how well the application server resources are used to serve requests. If not configured properly, a customer can suffer numerous service misses. It is the responsibility of the E-business system to prevent or minimize the possibility of violation of the agreement with its customers. Then it is significant to be able to predict and improve the performance of the Business Tier [62].

Therefore, the model we present in Chapter 4 will be primarily application server-centric, focusing on the operations at the business tier and interactions between the application server and the database.

3.4 Controlling the System Configuration

Another factor that we shall consider in analyzing the performance of an application server is the set of configuration parameters that influence the performance of application servers. In general application servers are designed to be flexible, in fact are provided with "knobs" or controllable parameters that can be adjusted in order vary the configuration setup and optimize the performance. In this section we provide an overview of these parameters and their influence on the performance.

Performance of application servers depends heavily on appropriate configuration, but to choose the correct settings of parameters is a difficult and error-prone task. The system administrator must configure the application server so that it can appropriately manage the concurrency or the building up of queues in a running application. In his work, the administrator has to consider the hardware and the software systems interacting with the application, as well as the workload that determines the aspects of the application will be used most heavily. For example, the configuration of the application server is different depending on whether the back-end database is dedicated or shared, as well as on whether the workload is made of few users with long and heavy sessions or many users with shorter sessions[40].

Generally, the administrator of the server nodes manually tunes these parameters to optimize the performance. There are best-practice guides[17] and ruled based profilers (based on the best-practices) that suggest configuration settings based on usage data. But best practice are heuristic and can not cover the entire space of applications, hardware/software structures and workloads. There are cases (see [49]) in which the best configuration of the server is found to be in conflict with the best practice suggestion. It happens then that in many cases application servers are configured using a mixture of rules-of-thumb, intuition and trial-and-error approaches.

Furthermore, there are hundreds of parameters that can be modified and many of them present complex interactions. Not all configuration parameters will be relevant for any given application, but determining the relevant parameters and setting them appropriately can have a significant effect on the performance of an application, and sometimes, even determines whether the application executes at all.

3.4.1 Static Versus Dynamic Control

An important distinction among the overall setting of configuration parameters is that the modifications can be performed in a *static* or *dynamic* manner. In the former case, parameters are assigned only at server down-time; in the latter case the parameters can be tuned at run-time.

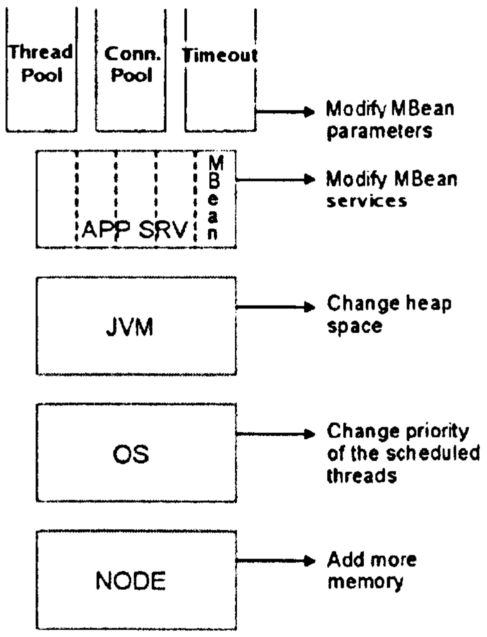


Figure 3.3: Varying the Configuration of an Application Server Node

In an application server node, the configuration can be modified at different levels, as illustrated in Figure 3.3. For instance, starting from the physical level we have the following:

1. At the hardware level more physical resources can be provided, increasing the physical memory or the speed of the processor.

2. The operating system is inherently designed to optimize the scheduling of threads,

however there is some control over this behavior. The priority of process threads can be modified to adjust for situations in which scheduling is unsatisfactory, although this could introduce the risk of deadlocks.

3. At the process level, the JVM can be started with the `Xmx <size>` option, which sets the maximum JVM heap size, which the maximum amount of memory allocated to the JVM in which the application server executes. For instance, to increase the memory for the JVM during our experiments, in the JBoss startup script we set the Java environment variable to `-server -Xms128m -Xmx512m`. However, providing more virtual memory at the JVM can improve performance, with the caveat that paging can badly affect performance if more memory is allocated than the physical memory.

4. At the server level, the group of services for the application server start-up can be specified: (a) the minimum number of services required to start the server, (c) the default services for J2EE, or (c) the configuration containing all the available services, included clustering. As we will see in Chapter 6, in JBoss those services are represented by MBeans components.

5. At the component level, examples include the configuration of multiple thread and connection pools, queues, cache size and timeout and retry values. This is the only control that can be dynamically performed, simply accessing the MBean components at server run-time.

In our work we mainly focus on analyzing the effects of controlling the parameters at point 5, those belonging to the component level. The reason for this is that our aim is to give support to system administrators for *dynamically* selecting the appropriate parameter setup that would attain the desired QoS level. This must be done without bringing down the application server system, which generally is an unacceptable option.

3.4.2 Examples of Dynamically Controllable Parameters

Clients send their request to connect to the application server. If the server is busy, the requests are inserted in a waiting queue, when the queue reaches the maximum capacity the request is rejected. The maximum queue length for incoming client requests to connect is the *Backlog* parameter.

The application server is a multithreaded program, whose threads concurrently process client requests. The *Thread Pool Size* parameter sets the size of the pool where live server threads are maintained, to reduce the overhead of creating a new thread.

Database accesses are performed by means of live connection to the database. A thread requiring a connection holds until the connection returns the data. The number of Connections that can be concurrently handled is a controllable parameter regulated by the *Database Connection Pool Size* parameter, which sets the size of the pool of database connections.

If there are more application server threads than available connections, a request may have to wait in a queue for a connection to become available. That queue is special, in that each request has a timeout interval, if the timeout expires, the request is dropped and the job is unsuccessfully terminated. *Timeout* is another controllable parameter.

The Thread Pool Size for instance is a critical parameter, because it dictates the concurrency level at the application server. A small number of threads may work well for providing good response time, but there is higher probability of rejecting client requests, leaving the server under-utilized; on the other hand, a high number of concurrent threads increases utilization but slows down response time.

Our experience in tuning this parameter during benchmarking tests [13] shows that, at a fixed load, increasing the number of threads leads to an improvement of both response time and throughput; but this is true until only up to a certain load, after which the application server saturates. The trade-off is that there is a point at which the overhead associated with context-switching, i.e. giving the CPU to each of the threads in turn, becomes so costly that performance dramatically degrades. Therefore it is important to find the best setting at a given workload, even applied to more than one parameter, which may influence each other.

One could argue that it is sufficient to size the system so that it can provide the best resource availability for serving the maximum number of requests under peak of loads. For instance, it could be possible to set Thread Pool Size at the highest value, even if the load is low. When the workload intensity increases, there will already be the maximum available number of threads waiting for serving a high number of incoming requests.

It happens instead that a balanced configuration setting can avoid build up queues of jobs waiting to be served. As shown in Figure 3.4, client requests can be awaiting either at the application server door, waiting to be served by a server thread, or at the database server door, waiting to write/read data in the database for its computations. If the number of threads serving requests is too high, the queue at the database increases and the database become congested. On the other hand a low number of threads can build up queues at the application server door, leaving the database under-utilized.

Another important reason for applying configuration control is that a shared resource is optimally accessed without setting the configuration parameter at its highest. For

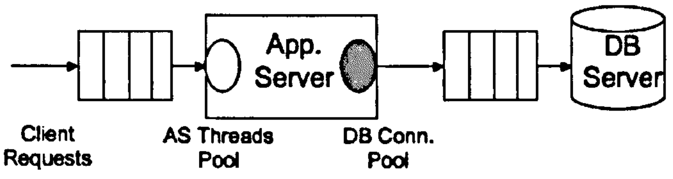


Figure 3.4: Queues at Different Tiers

example, we consider the case of a database shared among different application server instances in a cluster. Each application server maintains its pool of database connections and the size of the pool is the maximum number of connections to the database that can be open concurrently. It is worth keeping the number of connections at a low level when there is no high workload, so that the access on the connection table is faster for the overall set of nodes in the cluster.

As result, it is worth tuning the configuration parameters of an application server in order to support the achievement of the optimal system performance at any time, without leaving resources under-utilized.

3.5 Summary

In this chapter we analyze the the activities within an E-business system. It is worth noticing that the model captures the behaviour of a large class of E-business systems, since it is not tied to a specific technology. Although the typical request pattern of a J2EE application is used to exemplify the interactions, the activities are represented in a conceptual manner that abstracts from implementation details. And yet, the concepts are sufficiently detailed to take into account the essential system features.

We describe the different tiers that compose the system architecture, identifying the main functionalities of each tier. We focus our attention on the last two tiers, the business and data tier, since an E-business system profit depends on how well the application server resources (and the related persistent data) are used to serve requests. On the business tier we identify the various aspects that can influence the system behaviour, and also the main configuration parameters that can be modified, possibly at runtime, in order to control the performance.

The outcome is the analytical model that will be illustrated in Chapter 4, which allows evaluation of performance metrics in terms of measurable or controllable parameters.

Chapter 4

Queuing Network Model and Approximation

4.1 Introduction

In this chapter we define the analytical model and its approximate solution, designed to be numerically tractable and, at the same time, accurate.

From the analysis of the model characterized in Chapter 3 we derive the following remarks.

1. The work performed in the presentation tier is much less demanding than that carried out in the other two tiers, therefore a well-provisioned presentation tier is usually not a performance bottleneck (see [63]). Hence, in what follows, we will consider that the E-business system is made up of just the last two tiers, subjected to an external stream of requests.

2. The application server is a multithreaded program, the threads running here serve the request from the previous tier and perform computations using business logic and enterprise data. The data required by the computation has to be fetched from the database, in either a transactional or non-transactional manner.

3. The database access requests from the Business Tier are blocking in nature. That is, an application server thread waits until a database server returns the requested data or terminates the transaction. This means that the number of pending database access requests cannot exceed the number of application server threads.

4. If there are more application server threads than database servers, a request may have to wait in a queue for a database server to become available. That queue is special, in that each request has a timeout interval; the timeout is cancelled when the request

is taken up for processing by a database server; if and when the timeout expires, the request is dropped and the calling thread is freed, resulting in an unsuccessful termination for the user job.

5. Finally, the Data Tier is represented by a set of database "servers" which maintain the persistent data. In this context, the term "server' is used in the sense of "dedicated unit of database capacity", which may include processing power and disk storage; there is no connotation of a physical entity but there is an assumption that these units do not interfere with each other.

In the subsequent chapters, we use these remarks to describe the queuing network that constitute our model, from which we obtain an usable approximate solution.

4.2 The Model

4.2.1 Preliminaries and Assumptions

The model is expressed using two primitive data structures: *threads* and *queues*. The former process the client requests and each tier is modeled as a collection of threads of distinct type. The queues hold those requests that are awaiting to be processed by a thread.

A thread is in the *active* state while it is processing a request; as a part of processing, it may make a 'blocking' call to another thread and enter the *blocked* state. When the call returns, the caller thread re-enters the active state. Finally, a thread is in the *available* state when it is neither active nor blocked. When a request arrives at a tier, it is either handed to an available thread (if any) of that tier which subsequently becomes active, or kept in a queue until a thread becomes available to get active on it. A queue can thus be either *empty* or *non-empty*.

We assume that the threads are reliable, they complete processing of any request in some finite time, and a remote call always returns with a response that can be *normal* or *exceptional*. Thus, threads cannot remain forever in the *active* state over a given request and in the *blocked* state over a given call.

We consider a special class of queues, called the *Timed Queues*, characterized as follows. If an entry in a timed queue is not dequeued by a thread within Δ seconds, it is deleted and the thread that enqueued the deleted entry receives an exceptional response. Thus, a timed queue models a request waiting on a timeout for a resource to become available and an available resource being allocated to a waiting request in the FIFO manner. We note here that the notion of *queues with reneging* can be used to model and analyze the behaviour of requests in timed queues.

A queue, timed or otherwise, is assumed never to become full and an incoming request is always enqueued. The rationale for this assumption is two-fold.

1. We assume that an admission control policy is operative for requests entering the E-business system; an aspect of this policy is to admit requests only if the queue in the Presentation Tier is not full.

2. Since a remote call is blocking in nature, the number of requests awaiting to be processed in the second or the third tier cannot exceed the number of threads in the previous tier which is finite and often known. A careful provisioning of buffer space avoids queues at the Business and Data Tiers from becoming full.

For the reasons stated in Chapter 3, it is the performance of the *Business and Data Tiers* which crucially influence the system performance. So, the *System Model* is concerned only with these two tiers; the clients accessing these tiers are represented by *proxy clients* (typically the servlets of the Presentation Tier) which provide dynamic web pages to actual clients.

4.2.2 Description

The Business and Data Tiers are modelled by the (non-separable) queueing network illustrated in figure 4.1.

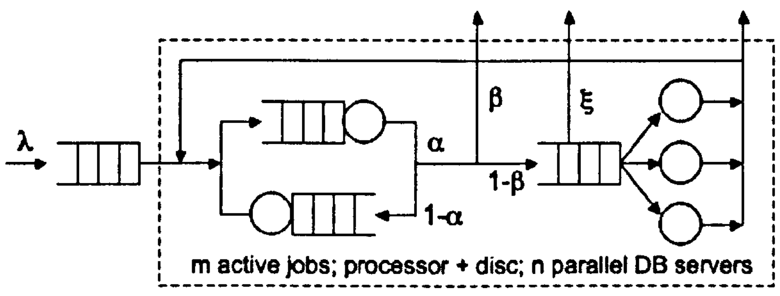


Figure 4.1: An Application Server with Database Access.

Requests (or 'jobs', as they will be called from now on) arrive into the system in a Poisson stream with rate λ. A maximum of m jobs are admitted for processing (they are referred to as the 'active' jobs); this limit is imposed by the number of parallel threads that have been made available. If there are more than m jobs present, the ones that do not occupy a thread wait in an external FIFO queue.

The execution of an active job consists of alternating 'computations' and 'database accesses'. Computation services are provided by a computer which, for the purposes of this model, is assumed to consist of one processor and one disc (the single disc assumption is not significant; one could easily generalize it by allowing multiple discs, at the price of complicating the solution). During a computation, jobs visit the processor at least once and the disc zero or more times. More precisely, after receiving a service at the processor, a job goes to the disc with probability $1-\alpha$ and completes the computation with probability α $(0 < \alpha \leq 1)$.This models a geometrically distributed number of visits to the disc. After a disc service, a job returns to the processor with probability 1. There is a FIFO queue at both the processor and the disc.

Service times at the processor and the disc are assumed to be independent random variables distributed exponentially with means b_0 and b_1, respectively.

Having completed a computation, a job terminates execution (and leaves the application server) with probability β, and makes a database access with probability $1 - \beta$. In other words, a job makes a geometrically distributed number of database requests. Database accesses are handled by a database engine, which may possibly be shared by more than one application server. As far as this application server is concerned, up to n of its active jobs may have their database accesses processed in parallel without significant interference. This is modelled by assuming that there are n parallel database servers. If more than n jobs need to access the database, n of them are being served while the others wait in a FIFO queue.

A time-out policy operates at the database engine: if, having asked for for a database access, a job does not acquire a database server before its time-out period expires, that job is terminated and leaves the application server. In practice, the lengths of the time-out periods are fixed; however, for purposes of analytical tractability, it is assumed that they are independent random variables distributed exponentially with mean $1/\xi$.

Database accesses are further complicated by the fact that they can be of two types. Some are read-only queries; their service times are distributed exponentially with mean b_2. Others are transactions which typically take longer; they are distributed exponentially with mean b_3. After the completion of a read-only query, a job enters a new computation phase. On the other hand, a transaction is always the last of a job's database accesses; whether it ends with a commit or a roll back, after it the job terminates and leaves the application server. A fraction θ of all jobs that make database accesses terminate with a transaction $(0 \leq \theta \leq 1)$.

When an active job leaves the application server, whether successfully or unsuccessfully, the job at the head of the external queue (if any), moves to the processor queue; i.e., it becomes active and starts a computation.

4.2.3 Remarks

1. When the external queue is non-empty, the application server behaves like a closed queueing network, with m jobs circulating between the computation and database engines. However, that network does not have a product-form solution (and hence is not tractable by mean-value analysis), because of the time-out departures from the database queue. That feature necessitates the approximate solution that will be presented in the next section.
2. The assumption that the computation part of the application server consists of a single processor and a single disc is not essential. One could envisage considerably more complex networks of processors and discs. An approximation would still be available, but would have to rely on numerical solution of equations; the closed-form expressions for the state-dependent computation throughput would not apply.

4.3 Approximate solution

Consider first the computation subsystem, with k jobs circulating between the processor and the disc ($k = 0, 1, \ldots, m$). Suppose that that circulation continues for a long time, i.e. the computation subsystem reaches steady-state with k jobs. Then the processor queue would behave like an M/M/1 queue with a bounded buffer of size k, into which jobs arrive at rate $1/b_1$ (when not full), and from which jobs depart at rate $(1 - \alpha)/b_0$ (when not empty). Denote by r the ratio between those two rates:

$$r = \frac{(1-\alpha)b_1}{b_0} . \tag{4.1}$$

Known results for the M/M/1/k queue yield the probability, U_k, that the processor is busy, given that k active jobs are in their computational phase (e.g., see [43]):

$$U_k = \frac{1 - r^k}{1 - r^{k+1}} . \tag{4.2}$$

Hence, the state-dependent throughput of the computation subsystem when there are k jobs in it, t_k, is given by

$$t_k = \frac{\alpha}{b_0} U_k = \frac{\alpha}{b_0} \frac{1 - r^k}{1 - r^{k+1}} . \tag{4.3}$$

So, the first approximation is to replace the computation subsystem by a single state-dependent server whose service rate, when there are k jobs present, is t_k.

Now suppose that there are J active jobs, circulating between the computing and database engines for a sufficiently long period to reach steady state ($J = 1, 2, \ldots, m$). In other words, model the business and data tiers by the closed queueing network illustrated in Figure 4.2

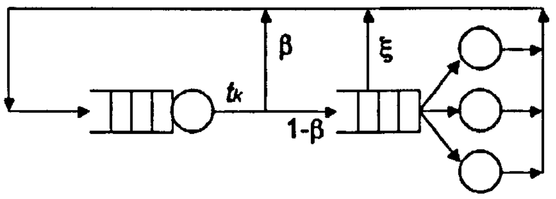


Figure 4.2: Closed Queueing Network Model.

The state of this network is described by a single integer, k, specifying how many jobs are at the computing node; then $J - k$ jobs are at the database node. A transition from state k to state $k + 1$ occurs when a job leaves the database node (through service completion or time-out); similarly, a transition from state $k + 1$ to state k occurs when a job leaves the computing node.

Let p_k be the probability that the network is in state k ($k = 0, 1, \ldots, J$). The above implies that these probabilities satisfy the following balance equations:

$$[\min(n, J - k)\nu + \max(0, J - n - k)\xi]p_k = (1 - \beta)t_{k+1}p_{k+1} \ , \tag{4.4}$$

where ν is the average service rate of a database server.

This is a simple set of recurrences, of the form $p_{k+1} = a_k p_k$. Together with the normalizing equation ($p_0 + p_1 + \ldots + p_J = 1$), and the expressions (4.3) for t_k, they are easily solved. However, the average service rate ν is yet to be determined.

Recall that there are two types of database services, read-only queries and transactions. Given that a job visits the database at least once, it makes i such visits with probability $(1 - \beta)^{i-1}\beta$. The last of those is a transaction with probability θ. Hence, the fraction, τ, of all visits to the database that are transactions, is given by

$$\tau = \theta\beta \sum_{i=1}^{\infty} \frac{1}{i}(1 - \beta)^{i-1} = \frac{\theta\beta}{1 - \beta} \sum_{i=1}^{\infty} \frac{1}{i}(1 - \beta)^{i} = \frac{\theta\beta}{1 - \beta} \ln \frac{1}{\beta} \ . \tag{4.5}$$

Since the average service times of read-only queries and transactions are b_2 and b_3, respectively, the overall average database service time is

$$\frac{1}{\nu} = (1 - \tau)b_2 + \tau b_3 \ , \tag{4.6}$$

with τ given by (4.5). A mixture of different exponentially distributed random variables is not, in general, exponentially distributed. Nevertheless, the approximate solution treats the database services as if they were distributed exponentially with mean $1/\nu$.

Having solved equations (4.4) and determined the probabilities p_k, the state-dependent throughput of the application server when it has J active jobs, $T(J)$, is obtained from

$$T(J) = \sum_{k=0}^{J} p_k[t_k\beta + \min(n, J-k)\nu\tau + \max(0, J-n-k)\xi] . \tag{4.7}$$

That throughput includes both successful and unsuccessful terminations.

The final approximation is to treat the closed queueing network in figure 4.2 as a single state-dependent server whose service rate is $T(J)$ when there are J active jobs. Now the external queue, together with the active jobs, behave like a one-dimensional Birth-and-Death process which is in state M if there is a total of M jobs present ($M = 0, 1, \ldots$). Then the number of active jobs is $\min(M, m)$ and the number of jobs in the external queue is $M - \min(M, m)$. There are transitions from state M to state $M+1$ with rate λ, and from state $M+1$ to state M with rate $T(\min(M+1, m))$.

Denote by q_M the steady-state probability that there are M jobs present. These probabilities are determined by the balance equations

$$\lambda q_M = T(\min(M+1, m))q_{M+1} , \quad M = 0, 1, \ldots , \tag{4.8}$$

and the normalizing equation

$$\sum_{M=0}^{\infty} q_M = 1 . \tag{4.9}$$

Note that the departure rate in the right-hand side of (4.8) ceases to depend on M when the latter exceeds m. Therefore, we have

$$q_{m+i} = \rho^i q_m , \quad i = 1, 2, \ldots , \tag{4.10}$$

where $\rho = \lambda/T(m)$ is the offered load when the number of active jobs is the maximum allowable. This leaves only a finite set of equations to solve.

The process is ergodic (i.e., the queue is stable), if $\lambda < T(m)$, or $\rho < 1$.

One could also assume a bounded external queue, with maximum size K. Then the possible states would be $M = 0, 1, \ldots, m + K$; equations (4.8) would apply only up to $M = m + K - 1$; the sum in the normalizing equation would be finite and the question of stability would not arise.

Given the steady-state probabilities q_M, one can compute certain performance measures. The average number of jobs in the system, L, is

$$L = \sum_{M=1}^{\infty} Mq_M = \sum_{M=1}^{m-1} Mq_M + \frac{q_m}{1-\rho}\left[m + \frac{\rho}{1-\rho}\right] . \tag{4.11}$$

In the case of a bounded external queue, with maximum size K, that expression becomes

$$L = \sum_{M=1}^{m-1} Mq_M + \frac{q_m(1-\rho^{K+1})}{1-\rho}\left[m + \frac{\rho}{1-\rho}\right] - \frac{Kq_m\rho^{K+1}}{1-\rho} . \tag{4.12}$$

The average system throughput, T, is equal to λ when the external queue is unbounded and stable. When it is bounded at level K, the throughput is

$$T = \lambda(1 - q_{m+K}) . \tag{4.13}$$

When the system throughput is not equal to λ (i.e., the external queue is bounded), some incoming jobs are rejected. In that case, the probability of rejection, R, is given by

$$R = \frac{\lambda - T}{\lambda} = q_{m+K} . \tag{4.14}$$

The average response time, W, is obtained from Little's result:

$$W = \frac{L}{T} . \tag{4.15}$$

Note that some of the system throughput is in fact due to database requests that have timed out, or to transactions that have rolled back. Hence, one may be interested in the 'successful throughput', S, defined as the average number of jobs that complete successfully per unit time. Denote also the 'state-dependent successful throughput', given that there are J active jobs, by $S(J)$. That quantity is given by an expression similar to (4.7), but excluding the unsuccessful terminations:

$$S(J) = \sum_{k=0}^{J} p_k[t_k\beta + \min(n, J-k)\nu_T(1-\gamma)] , \tag{4.16}$$

where γ is the probability that a transaction fails to commit (that is a database parameter).

Then, the unconditional successful throughput is obtained from

$$S = \sum_{M=0}^{m-1} q_M S(M) + S(m) \sum_{M=m}^{\infty} q_M , \tag{4.17}$$

where q_M is the solution of (4.8) and (4.9).

4.3.1 Modified Approximate Solution

The modifications at the approximate solution described in this section will be applied only during the calibration process, described in Chapter 7. In order to isolate the performance of the application server only, let us consider the case when there is no database tier. In other words the system is determined only by the closed queuing network illustrated in Figure 4.3.

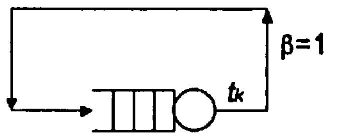


Figure 4.3: Application Server Only.

The observed data are collected at the steady state, when all the application server threads are busy serving requests. As previously stated, this M/M/1 queue represents the approximation of the computation subsystem at the steady state, when there are k jobs in it. The state-dependent throughput of the computation subsystem, t_k, is given by equation (4.3)

$$t_k = \frac{\alpha}{b_0} U_k \, . \tag{4.18}$$

where b_0 is the service time at the CPU per job per visit, and α is the probability of leaving the subsystem instead of circulating any longer between CPU and Disc. The average number of cycles a job makes in the subsystem, in other words the number of visits a job makes at the CPU, is then equal to $1/\alpha$. Then the average service time at CPU per job, C, is b_0 times number of visits, or, if we observe the system for a certain period Δ, the total CPU time b_Δ consumed in that period divided by the total number of job completed j_Δ .

$$C = b_0 \frac{1}{\alpha} = \frac{b_\Delta}{j_\Delta} \, . \tag{4.19}$$

U_k in equation (4.18) denotes the probability that the processor is busy given that k active jobs are in their computational phase. In other words U_k is the fraction of time the processor is busy, which is the total CPU time consumed b_Δ in the observation period Δ (i.e. the CPU utilization). Then (4.2) is:

$$U_k = \frac{1 - r^k}{1 - r^{k+1}} = \frac{b_\Delta}{\Delta} \, . \tag{4.20}$$

The equation (4.18) becomes:

$$t_k = \frac{\alpha}{b_0} U_k = \frac{1}{C} U_k \, . \tag{4.21}$$

which is the Utilization Law. Substituting C and U_k, we have:

$$t = \frac{j_\Delta}{b_\Delta} \frac{b_\Delta}{\Delta} = \frac{j_\Delta}{\Delta} \, . \tag{4.22}$$

This equation represents the observed service rate of the computation subsystem in the observed period Δ.

At the steady state, the average number of jobs in the system L is:

$$L = m. \tag{4.23}$$

there where all the m server threads are busy serving a job. In case of a bounded external queue, with maximum size K, that expression becomes

$$L = m + K. \tag{4.24}$$

The average system throughput, t, is equal to λ when the external queue is unbounded and stable. When it is bounded at level K, the throughput is

$$T = \lambda(1 - t) \, . \tag{4.25}$$

When the system throughput is not equal to λ (i.e., the external queue is bounded), some incoming jobs are rejected. In that case, the probability of rejection, R, is given by

$$R = \frac{\lambda - T}{\lambda} = t \, . \tag{4.26}$$

The average response time, W, is obtained from Little's result, as shown in Equation 4.15.

In this paragraph we have described the particular case of no database access ($\beta = 1$) for calibration purposes, in order to isolate the performance of the application server and easily compare the observed and approximated parameters. However the same formulas can be applied to the extended system, composed of both the application server and the database and $0 < \beta < 1$.

4.3.2 Approximating U_k

It is worth noting that, in equation (4.18), the value U_k does depend on the number of jobs in the subsystem, k, whereas $\frac{a}{b_0}$ does not. Therefore U_k is state-dependant. We further analyze this value in order to obtain a better estimate than the observation over the overall testing phase period.

U_k can be estimated either making an average of multiple observations of the ratio $\frac{b_\Delta}{\Delta}$ at smaller observation periods, or finding a solution for r in the equation

$$U_k = \frac{1 - r^k}{1 - r^{k+1}} . \tag{4.27}$$

In the latter case, we have to solve with an iterative method the formula, reduced in the form of

$$r = cUr^{\frac{1}{k}} . \tag{4.28}$$

using a given observation for U and k and an initial guess for r_0. r_0 will be iteratively substituted in the formula (4.28) for a significant number of iterations or until r_{i+1} is close enough to r_i. The initial guess r_0 is correct if the values converge.

4.4 Summary and Contributions

In this chapter we introduce the analytical model that describes the behaviour of an E-business system comprised of an application server and a database server. The contribution of our work lies in the incorporation into the model of *all* of the following features:

1. A user job may involve a random number of accesses to a database, each access being either read-only (lightweight) or transactional (heavyweight). A heavyweight access may terminate successfully (commit) or unsuccessfully (roll back).

2. The database itself may be shared among more than one E-business system. Hence, the database processing capacity (modelled as a number of parallel servers) allocated to a particular system is a controllable operational parameter.

3. Database access requests are subject to time-out intervals: if a request is not granted access before its time-out interval expires, the job that generated it is ejected. This is common in commercial database servers [33].

4. Computational and database resources are possessed simultaneously; that is, while a request waits for and receives a database service, the job that generated it continues to occupy an application server thread.

5. The physical components of the application server (central processor, one or more discs) are modelled explicitly.
6. There is an arrival process of jobs; those that are not admitted for processing wait in an external queue of finite or infinite capacity; if the former, and the queue is full when a job arrives, that job is lost. The presence of that external queue implies that the system model is not, in general, a closed queueing network.

Some of the above features are taken into account in other studies illustrated in Chapter 2 (e.g., [38] handle 4, 5 and 6; [36] include a form of 4; [29] allow different types of requests, which may include 1). However, as far as we are aware, the entire collection is considered here for the first time.

The price paid for the generality of the model is that it cannot be solved exactly. The aim of the analysis is therefore to obtain a useable and efficient approximate solution. The accuracy of this solution will be evaluated in Chapter 5 by comparison with simulations.

Chapter 5

Validation of Approximation Through Simulation

5.1 Introduction

Through analysis of the model, presented in Chapter 3, we have obtained a numerically efficient approximate solution, as it is illustrated in Chapter 4. In this chapter we present the results gained by validating the approximate solution via Monte Carlo method, employing a simulation program with variable time increments.

The purpose of simulations is to evaluate the accuracy of the approximate solution by means of a series of numerical and simulation experiments. Therefore the graphs presented here compare the behaviour of both the simulated system with the approximate solution over several system configurations, under different load conditions.

The system configuration is varied by: (a) tuning the *application server Thread Pool Size* parameter, (b) tuning the *Database Connection Pool Size* parameter, (c) reducing the average service time at the CPU, and (d) tuning the *External Queue Size* parameter.

In all these cases, the graphs show that the approximation provides a reasonably accurate estimation. The analytical solution can be employed to predict the performance of a system under given or assumed loading conditions, and to choose optimal settings for certain controllable parameters with respect to specified performance measures.

5.2 Simulation Results

The accuracy of the approximate solution is evaluated by comparing the model estimates to those obtained from simulations. Simulation experiments are conducted to examine the performance of the system using different parameter settings and under various load conditions.

As the number of parameters is quite large, some of them were kept fixed throughout:

- Probability of visiting the disc ($1 - \alpha = 0.6$)
- Probability of requiring a database access ($1 - \beta = 0.6$)
- Fraction of database accesses containing a transaction ($\theta = 0.2$)
- Average service time for read-only access ($b_2 = 20$ msec)
- Average service time for transactions ($b_3 = 50$ msec)
- Database time-out interval ($1/\xi = 5$ sec), except in Figure 5.13
- Bound on external queue size ($K = 50$), except in Figure 5.11 and 5.12

The main parameters that are varied are those *dynamically controllable*, in other words the parameters that can be tuned at server run-time, as specified in Section 3.4. These parameters are: the *Database Connection Pool Size* parameter, i.e. the number of parallel database servers (N), which sets the size of the pool of live database connections that can be concurrently handled or reused in order to reduce the overhead of opening new connections; the *application server Thread Pool Size* parameter, i.e. the number of parallel threads (M), which represents the number of live server threads maintained in a pool accepting requests from the waiting queue and handling simultaneous client sessions. In particular, the last one is a critical parameter, because it dictates the concurrency level at the application server, in fact a small number of threads may work well for providing good response time, but there is higher probability of rejecting client requests, leaving the server under-utilized; on the contrary, a high number of concurrent threads increases utilization but slows down response time. This is actually documented with the first set of experiments.

5.2.1 Workload Levels

In our experiments we consider various workload levels, however in the graphs that follow we only show the most significant ones: (a) the arrival rate that determines the saturation point, when the system capabilities are fully stressed, (b) the arrival rate lower and (c) the arrival rate higher than the saturation point.

The saturation point depends on the characteristics of the system, therefore in the first two sets of experiments the analyzed arrival rates are (a) $\lambda = 0.037$, (b) $\lambda = 0.03$ and (c) $\lambda = 0.05$. In the third set of experiments instead, in order to simulate a faster application server, we observe a the system with an extreme configuration. In this case the arrival rates that we consider are (a) $\lambda = 0.6$ (b) $\lambda = 0.4$ (c) $\lambda = 0.8$.

As we observe in our experiments, the arrival rate at point (a) represents a sort of " turning point" between the analytical model and the simulation, there were the model estimates adhere to a smaller degree to the simulation results. The reason for this is that, as we discuss in the next sections, the approximation result tends to be pessimistic, therefore it underestimates the maximum achievable throughput. As consequence the analytical model reaches the saturation point earlier than the simulation, showing slightly degrading performance while, at the same point, the simulated system copes well with the workload.

5.2.2 Tuning the Thread Pool Size Parameter

With the first set of experiments we aim to quantify the effect of the number of threads on the performance of the system.

In Figure 5.1, the average response time is plotted against M for different values of the external arrival rate, λ. The figure shows a marked initial improvement in performance as the number of active jobs increases; the servers are being utilized more efficiently by being able to work on different jobs at the same time.

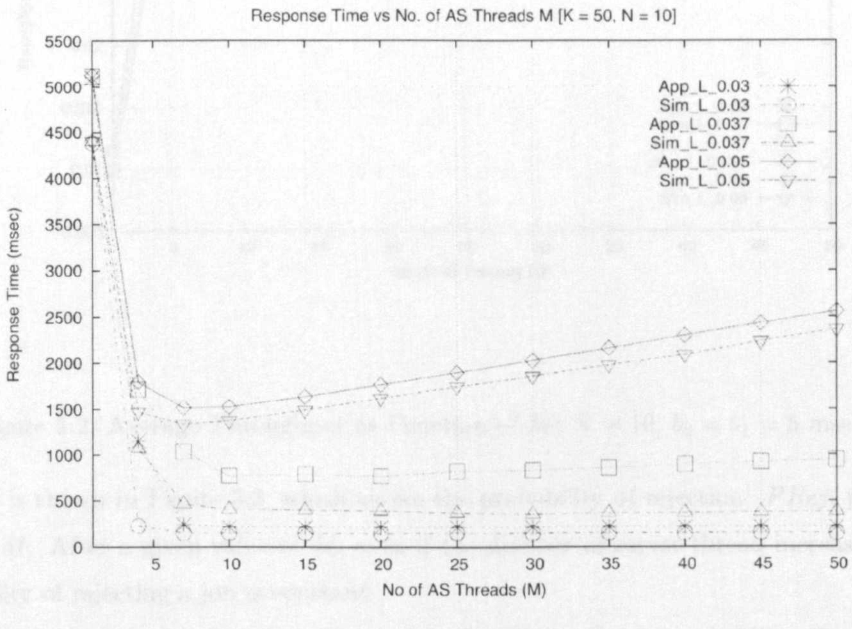


Figure 5.1: Average Response Time as Function of M; $N = 10$, $b_0 = b_1 = 5$ msec

However, the downward trend stops, or is reversed, when M increases further. That behaviour is readily explained by looking at Figure 5.2, where the throughput, T, is plotted against M for the same arrival rates. When all servers are busy most of the time, the throughput reaches a maximum achievable value, T_{max}. In the present configuration, that value is approximately 0.04 jobs/msec (or 40 jobs/sec). If the external arrival rate is lower than T_{max}, the throughput flattens before reaching that maximum. Further increases in M make no difference to the average number of jobs present and the average response time; the extra threads remain unused.

On the contrary, if $\lambda \geq T_{max}$, then the application server cannot cope with the demand and adding more threads increases the response time because it increases the concurrency within the application server.

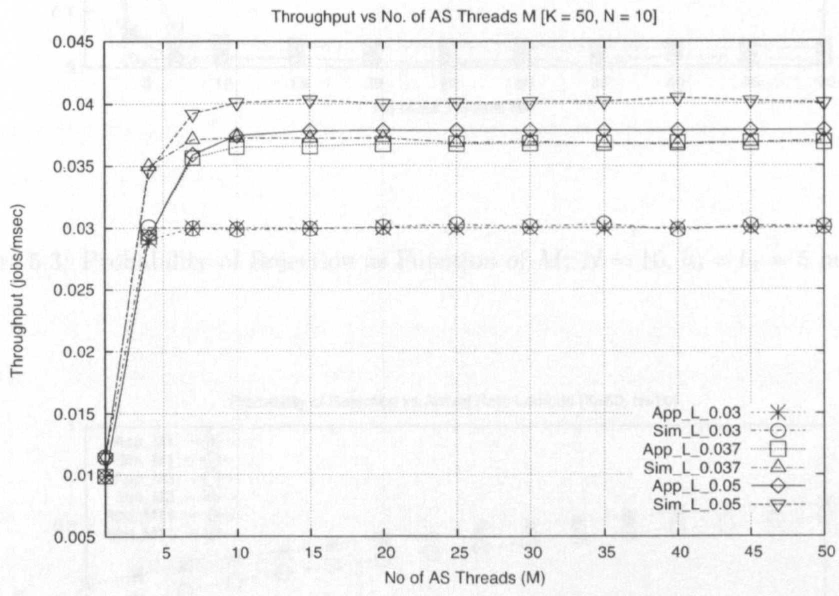


Figure 5.2: Average Throughput as Function of M; $N = 10$, $b_0 = b_1 = 5$ msec

This is shown in Figure 5.3, which shows the probability of rejection. $PRej$, plotted against M. After a given value of M, even if the number of server thread increases, the probability of rejecting a job is constant.

This behaviour is shown again in Figure 5.4, that illustrates the probability of rejection plotted against the arrival rate, for different numbers of threads. On the one hand, the rejection probability increases with λ because the system becomes more heavily loaded and hence the external queue is more often full. On the other hand, increasing M leads to a reduction in the rejection probability because more jobs can be admitted into the

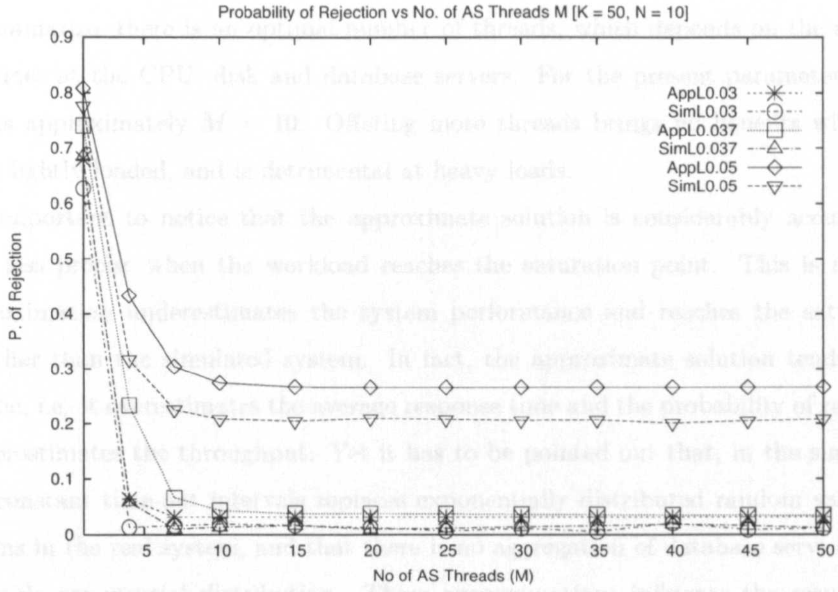


Figure 5.3: Probability of Rejection as Function of M; $N = 10$, $b_0 = b_1 = 5$ msec

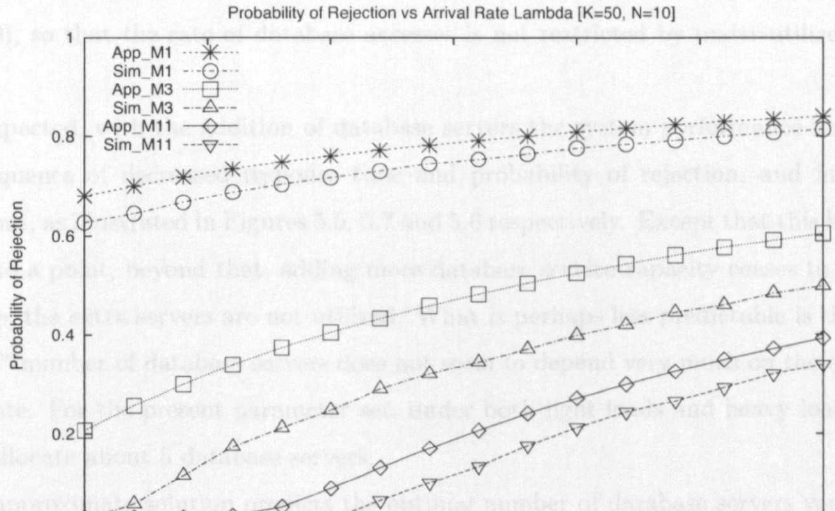


Figure 5.4: Probability of Rejection as Function of λ; $N = 10$, $b_0 = b_1 = 5$ msec

system. As consequence, the average number of jobs present increases and hence the average response time increases.

To summarize, there is an optimal number of threads, which depends on the average service times at the CPU, disk and database servers. For the present parameters, that number is approximately $M = 10$. Offering more threads brings no benefits when the system is lightly loaded, and is detrimental at heavy loads.

It is important to notice that the approximate solution is considerably accurate, it becomes less precise when the workload reaches the saturation point. This is because the approximation underestimates the system performance and reaches the saturation point earlier than the simulated system. In fact, the approximate solution tends to be pessimistic, i.e. it overestimates the average response time and the probability of rejection and underestimates the throughput. Yet it has to be pointed out that, in the simulated system, constant time-out intervals replaces exponentially distributed random variables, as happens in the real system, and that there is no aggregation of database service times into a single exponential distribution. These approximations influence the results and cause the slight underestimation.

5.2.3 Tuning the Database Pool Size Parameter

In the second set of experiments, the number of threads is kept fixed, while the number of available database servers is varied. The value of M is deliberately set quite high ($M = 50$), so that the rate of database accesses is not restricted by under-utilized CPU and disc.

As expected, with the addition of database servers the system performance improves, as consequence of decreased response time and probability of rejection, and increased throughput, as illustrated in Figures 5.5, 5.7 and 5.6 respectively. Except that this happens only up to a point, beyond that, adding more database service capacity ceases to make a difference; the extra servers are not utilized. What is perhaps less predictable is that this "optimal" number of database servers does not seem to depend very much on the external arrival rate. For the present parameter set, under both light loads and heavy loads, it is best to allocate about 5 database servers.

The approximate solution predicts the optimal number of database servers very accurately. And again, the model predictions are pessimistic, compared with the simulations. This phenomenon is observed in all experiments.

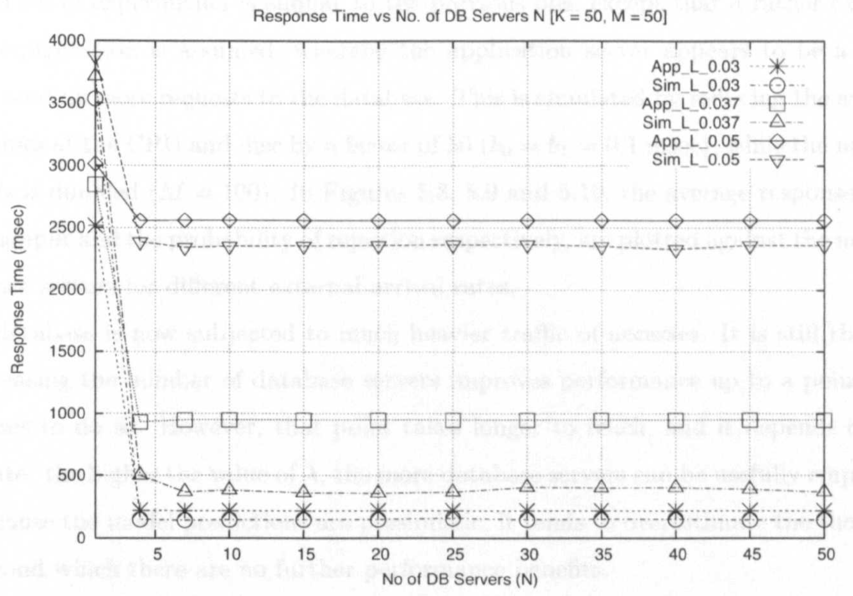


Figure 5.5: Average Response Time as Function of N; $M = 50$, $b_0 = b_1 = 5$ msec

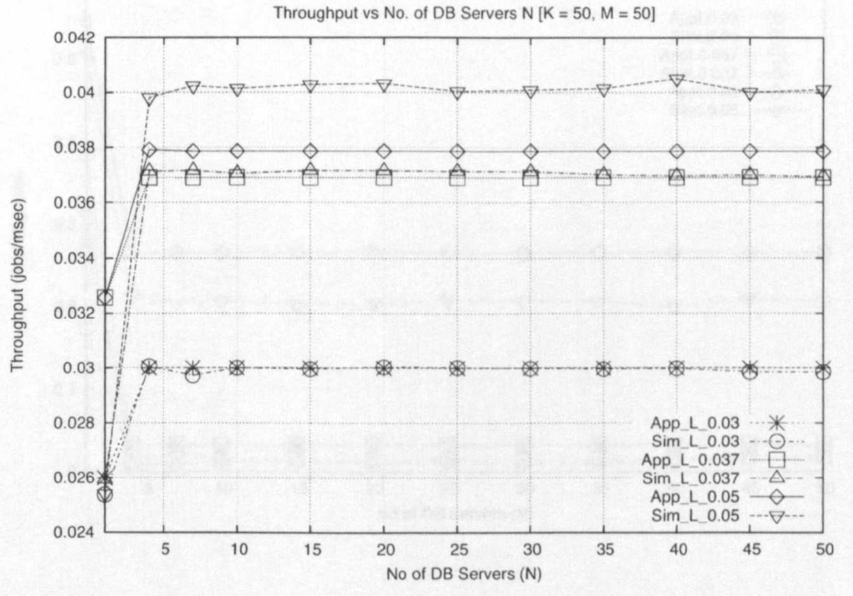


Figure 5.6: Average Throughput as Function of N; $M = 50$, $b_0 = b_1 = 5$ msec

5.2.4 Extreme Service Configuration

The third set of experiments is similar to the previous one, except that a rather extreme service configuration is assumed, whereby the application server appears to be a faster machine sending more requests to the database. This is simulated by reducing the average service times at the CPU and disc by a factor of 50 ($b_0 = b_1 = 0.1$ msec), while the number of threads is doubled ($M = 100$). In Figures 5.8, 5.9 and 5.10, the average response time, the throughput and the probability of rejection respectively, are plotted against the number of database servers for different external arrival rates.

The database is now subjected to much heavier traffic of accesses. It is still the case that increasing the number of database servers improves performance up to a point, and then ceases to do so. However, that point takes longer to reach, and it depends on the arrival rate: the higher the value of λ, the more database servers can be usefully employed. Also, because the model predictions are pessimistic, it tends to overestimate the the value of N beyond which there are no further performance benefits.

It is worth noticing that again the model estimates are reasonably accurate in all the figures.

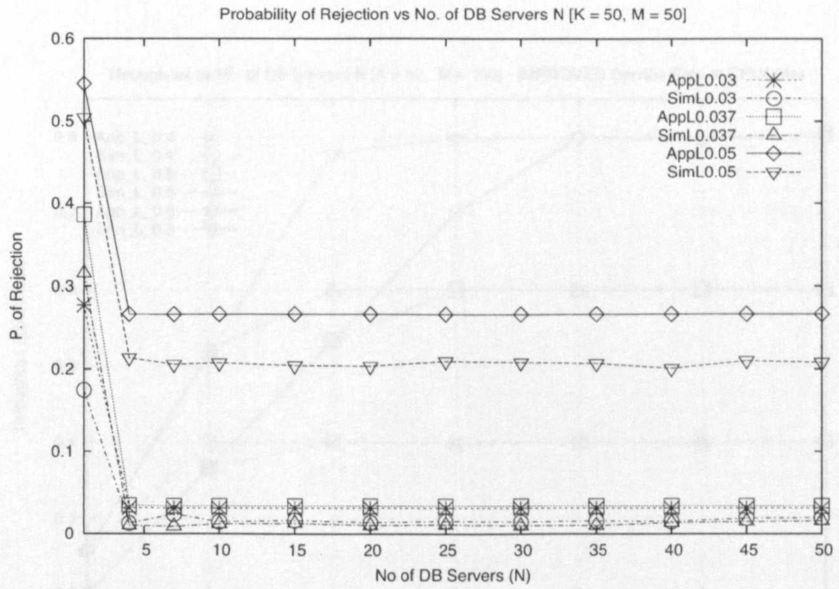


Figure 5.7: Probability of Rejection as Function of N; $M = 50$, $b_0 = b_1 = 5$ msec

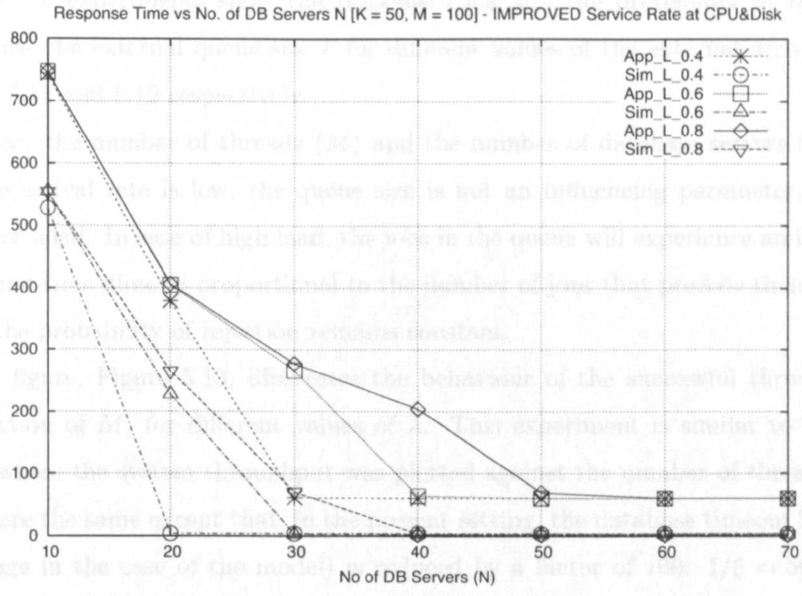


Figure 5.8: Average Response Time as Function of N; $M = 100$, $b_0 = b_1 = 0.1$ msec

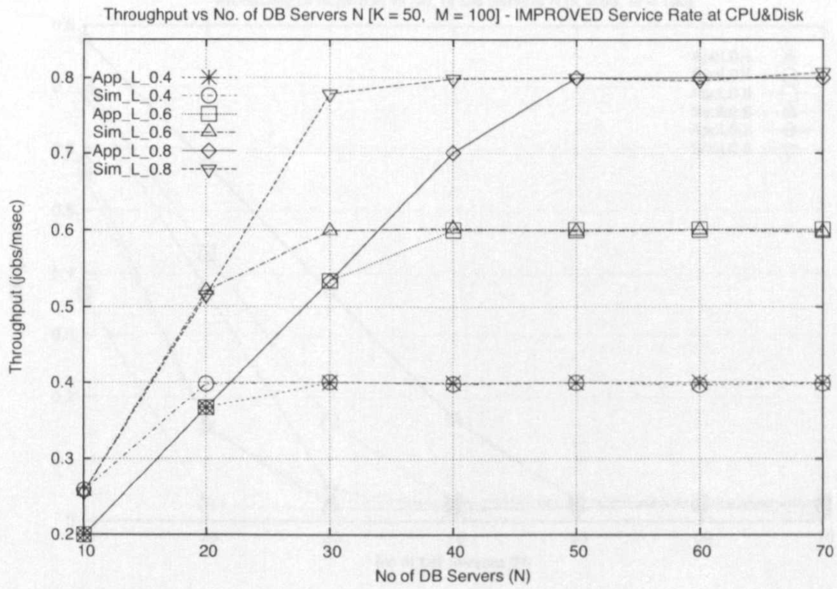


Figure 5.9: Average Throughput as Function of N; $M = 100$, $b_0 = b_1 = 0.1$ msec

5.2.5 Tuning the External Queue Size Parameter

A further set of experiments show the response time and the probability of rejection plotted against the external queue size k for different values of the external arrival rate, λ, in Figure 5.11 and 5.12 respectively.

In this case the number of threads (M) and the number of database servers (N) are fixed. If the arrival rate is low, the queue size is not an influencing parameter, as the queue is never filled. In case of high load, the jobs in the queue will experience an increase in the response time directly proportional to the number of jobs that precede them in the queue, but the probability of rejection remains constant.

The last figure, Figure 5.13, illustrates the behaviour of the successful throughput, S, as a function of M, for different values of λ. This experiment is similar to that in Figure 5.2, where the system throughput was plotted against the number of threads. All parameters are the same except that, in the present setting, the database timeout interval (or its average in the case of the model) is reduced by a factor of 100: $1/\xi = 50$ msec. Consequently, jobs are now more likely to be ejected from the application server because their timeouts expire.

Comparing the results in Figure 5.13 with those in Figure 5.2, we observe that at

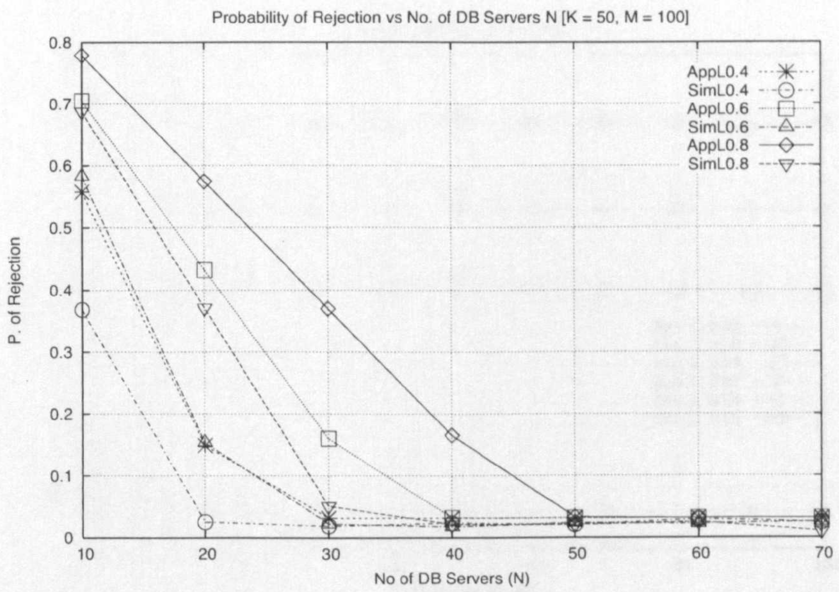


Figure 5.10: Probability of Rejection as Function of N; $M = 100$, $b_0 = b_1 = 0.1$ msec

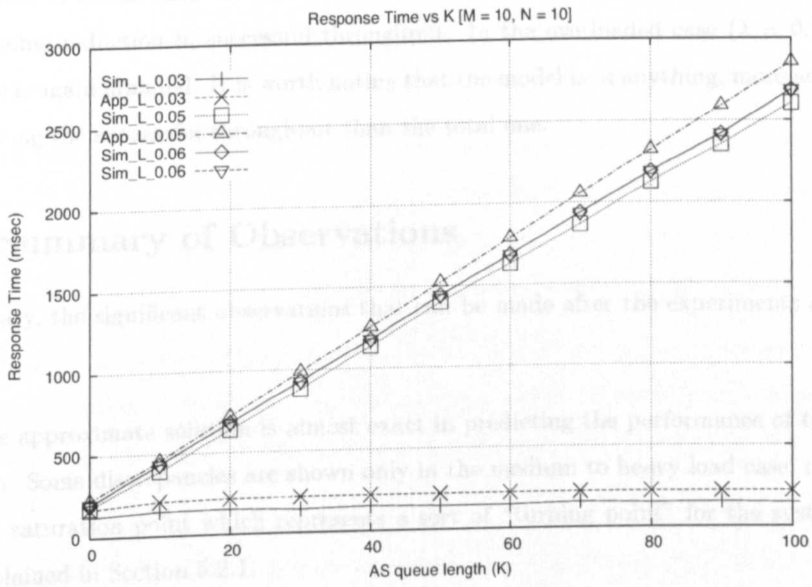


Figure 5.11: Response Time as Function of k; $N = 10$, $N = 10$, $b_0 = b_1 = 5$ msec

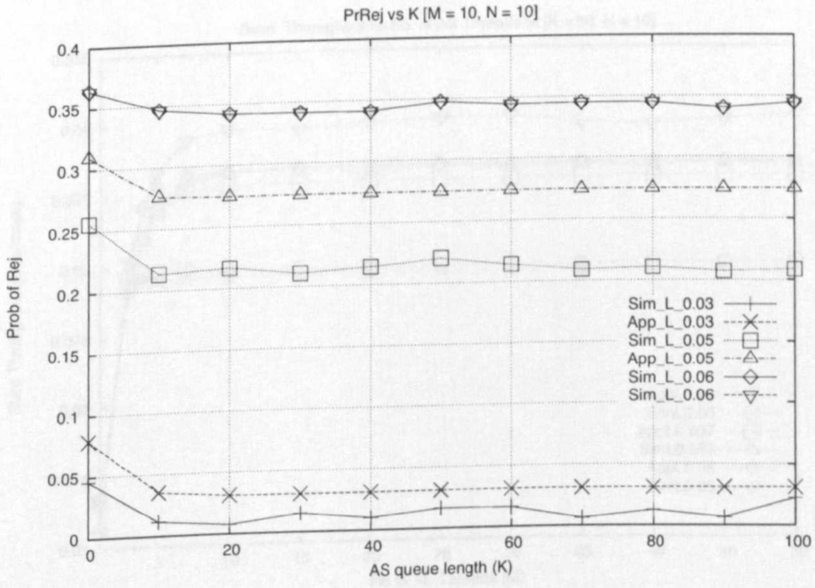


Figure 5.12: Probability of Rejection as Function of k; $N = 10$, $N = 10$, $b_0 = b_1 = 5$ msec

light load ($\lambda = 0.3$) the shorter timeout interval has made very little difference to the throughput: it is still almost entirely successful. At medium to heavy load ($\lambda = 0.37$), there is some reduction in successful throughput. In the overloaded case ($\lambda = 0.5$), the difference is again minimal. It is worth noting that the model is, if anything, more accurate in predicting the successful throughput than the total one.

5.3 Summary of Observations

In summary, the significant observations that can be made after the experiments are the following:

- The approximate solution is almost exact in predicting the performance of the system. Some discrepancies are shown only in the medium to heavy load case, close to the saturation point which represents a sort of "turning point" for the system, as explained in Section 5.2.1.
- The approximate solution tends to be pessimistic, i.e., it overestimates the average response time and underestimates the throughput. That is probably due to the fact that replacing the constant time-out intervals with exponentially distributed ones

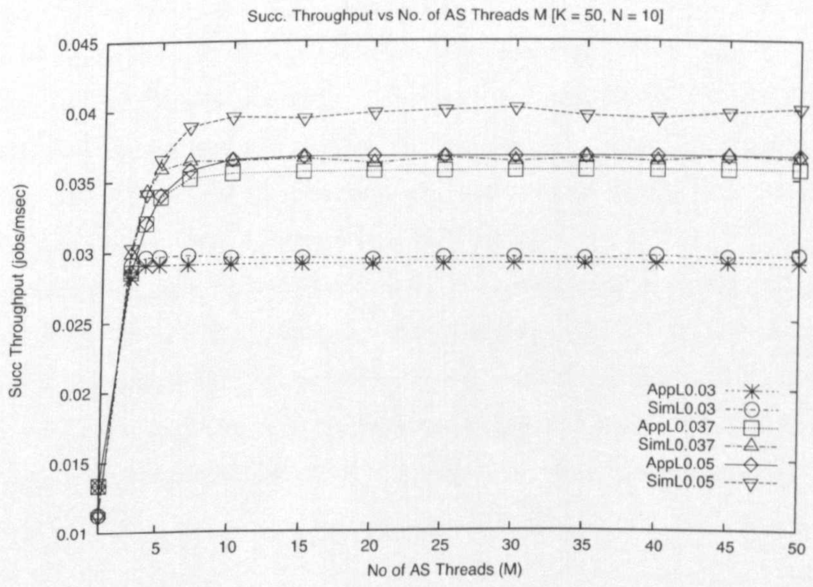


Figure 5.13: Successful Throughput as Function of M; $N = 10$, $b_0 = b_1 = 5$ msec

increases their variance and hence has an adverse effect on performance. On the contrary, aggregating the different database service times into a single exponential distribution has the opposite, but this is a less pronounced effect. However the model is more accurate in predicting the successful throughput than the total one.

- From the performance point of view, we have seen that there exists an optimal number of threads, which depends on the average service times at the CPU, disk and database servers. Further increases in M make no difference to the average number of jobs present and the average response time, while the extra threads remain unused.

- Similarly, the system performance improves and the throughput increases with the addition of database servers, but only up to a point. Beyond that, adding more database service capacity ceases to make a difference and the extra servers are not utilized.

As we will see in Chapter 7 the same considerations hold when comparing the approximate solution to a real world system of realistic size and complexity.

Chapter 6

Design and Implementation of the QoS Control System

6.1 Introduction

This chapter discusses our implementation of a QoS Control System that is capable of monitoring application server performance, as well as modifying its configuration parameters towards some QoS objectives. The QoS Control System is designed to be fully integrated within the JBoss application server, thus extending its functionalities. The JBoss application server, in combination with the Oracle database, is referred as the *Target System.*

The monitoring function is provided by *sensors*, components strategically placed within the Target System to monitor the system activities. The configuration parameters are modified by means of an *actuator* component, which can dynamically reconfigure the system at runtime and consequently alter its performance. Determining which configuration parameters need to be tuned and to what values is the task performed by the *QoS Controller Unit* (QoS-CU). These three components constitute the QoS Control System, their design and development will be discussed in this chapter.

The chapter is organized as follows. We begin with presenting an overview of the J2EE technology, in particular, we illustrate the main features of the JBoss application server. The overview characterizes the implementation environment and also describes the actual activities within the Target System, confirming the concepts analyzed in Chapter 3. We then proceed to discuss the implementation details of the three components that form the QoS Control System.

6.2 J2EE Technology Overview

As illustrated in Chapter 3, E-business applications are hosted by multi-tier architectures developed using component-oriented technologies. One of the predominant technologies employed in developing E-business applications is Suns *Java 2 Enterprise Edition (J2EE)* [55].

The primary aim of J2EE is to allow application developers to deploy distributed components in an environment where the middleware infrastructure provides essential support for aspects such as concurrency, persistence, transactions, database interactions and security. The application server is the infrastructure that supplies these functionalities, thus managing the environment in which business logic is executed.

The application server is a multi-threaded program running in a single process and supported by a single JVM. It provides various software components that aim to ease the development of complex E-business applications and it is designed to offer a scalable, high-performance infrastructure for processing many simultaneous requests.

The core components that form the application business logic are implemented using *Enterprise Java Beans* (EJB) [54]. Every bean has two types of interface: a *Component interface* that defines the business methods callable by the clients, and a *Home interface* that defines the methods used to create, remove and find a specific EJB.

The beans are deployed in a *container* that provides the run-time environment and makes the middleware services available. Among the other functionalities, the container mediates client/bean interactions, provides the naming service to locate the bean, supports remote communication, coordinates the transactions and synchronizes the bean state with the persistence storage. There are three types of EJBs:

Entity Beans define the persistent data of an application. An instance of such a bean represents a row of data in the database table. The bean is *activated* when the state of a bean is loaded into volatile memory, so that the related data can be manipulated. At the end of the operation, the bean state is saved in the database, making it persistent. The persistence management of the bean can be of two types: *Bean Managed Persistence* (BMP) and *Container Managed Persistence* (CMP). With BMP the synchronization of bean cached state with underlying database must be managed explicitly in the bean code, while with CMP, the container becomes solely responsible for that. Entity beans may be shared by multiple clients. It is therefore important that the beans work within transactions, since the clients might want to change the same data. Typically, during a transaction the entity beans are managed by the EJB container.

Session Beans are instantiated on a per-client basis, a given instance is available for use by only one client. These beans are used to perform work for the clients and encapsulate the business logic. When the client has finished its work, the client session terminates and the bean is no longer associated with the client. There are two types of session beans: *stateful* and *stateless*. The stateful bean is allocated to a specific client, whose requests are always routed to the same bean, maintaining the session state within the method invocations till the client completes. The stateless bean is not specific to a particular client; it contains a state only for the duration of a method invocation, when the method is finished, the state is no longer retained.

Message-driven Beans provide asynchronous processing by acting as message-listeners for *Java Message Service* (JMS) [57]. Clients do not contact these beans directly, but rather send messages to a given JMS queue for which the bean acts as a message consumer.

For the scope of this thesis, we focus on a particular J2EE implementation of application server, called *JBoss* [18], one of the most popular open-source application servers. Specifically, we used JBoss 3.2.3

6.3 JBoss Application Server

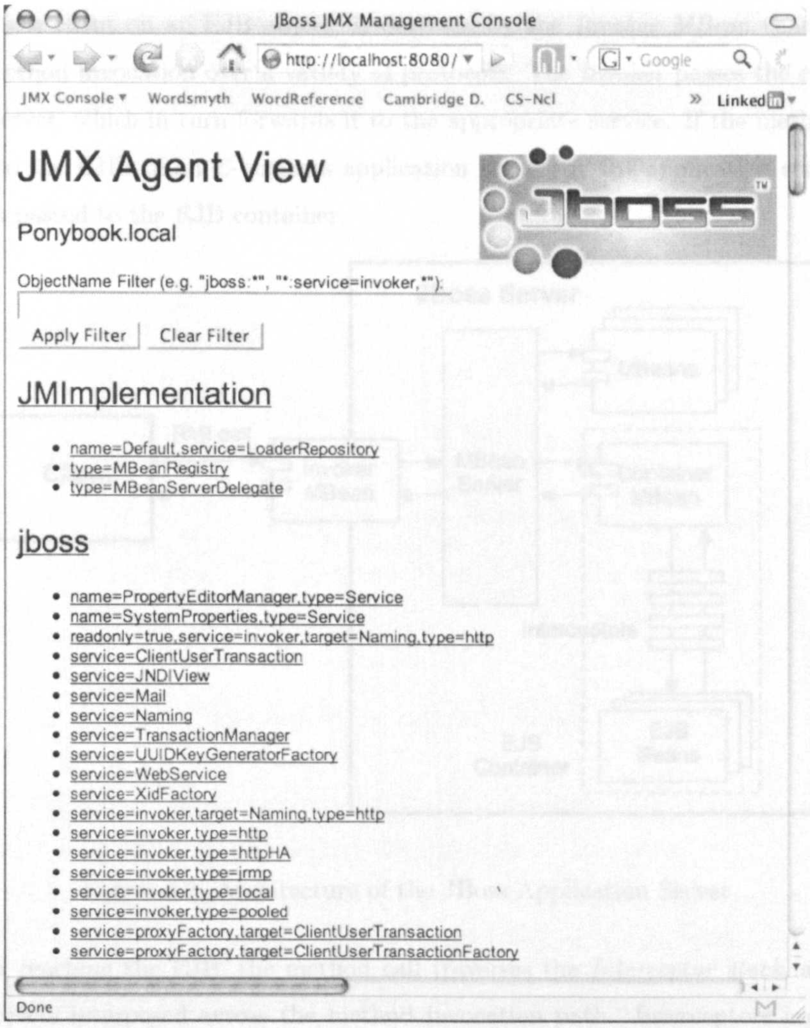
JBoss is an open-ended middleware, in the sense that users can extend middleware services by easily deploying new components into a running server [14]. This extensibility is important for us because it is exploited for instrumenting the application server with the QoS Control System.

The foundation of JBoss extensibility is the *Java Management Extension* (JMX) specification [32]. The JMX infrastructure provides JBoss with a lightweight environment where components can be dynamically loaded and updated.

JMX defines a common software bus, the *MBean Server*, that allows integration and centralized management of components defined as *managed beans (MBeans)*. MBeans act as wrappers for applications, components or resources in a distributed architecture.

In JBoss most of the key services of the J2EE architecture are implemented as MBeans, and also the new modules can be easily integrated as MBeans in the server. While, in contrast, other commercial application servers have a monolithic approach whereby the integration of new components can be troublesome.

The MBean Server represents the registry for MBeans that makes the MBean management interface available for use by other components or end-users. A JMX HTML adaptor



JMX Agent View

Ponybook.local

ObjectName Filter (e.g. "jboss:*", "*:service=invoker,*"):

JMImplementation

- name=Default,service=LoaderRepository
- type=MBeanRegistry
- type=MBeanServerDelegate

jboss

- name=PropertyEditorManager,type=Service
- name=SystemProperties,type=Service
- readonly=true,service=invoker,target=Naming,type=http
- service=ClientUserTransaction
- service=JNDIView
- service=Mail
- service=Naming
- service=TransactionManager
- service=UUIDKeyGeneratorFactory
- service=WebService
- service=XidFactory
- service=invoker,target=Naming,type=http
- service=invoker,type=http
- service=invoker,type=httpHA
- service=invoker,type=jrmp
- service=invoker,type=local
- service=invoker,type=pooled
- service=proxyFactory,target=ClientUserTransaction
- service=proxyFactory,target=ClientUserTransactionFactory

Figure 6.1: JMX Agent View in JBoss

allows the end-user to view the MBean Servers MBeans using a standard web browser. The default URL for the console web application is `http://localhost:8080/jmx-console`. The page that occurs at this location is presented in Figure 6.1. The top view is the *Agent View* that provides a listing of all MBeans registered with the MBean Server sorted by the domain portion of the MBean Object Name. Selecting one of the MBeans navigates to the related MBean view, where it is possible to examine and edit MBeans attributes as well as invoke operations.

The EJB container itself, where the beans of an E-business application are hosted, is also implemented as a MBean and contains some configuration aspects that can be selected and changed by users. The basic architecture of the JBoss application server is illustrated in Figure 6.2, where the execution of a method invocation is exemplified. The method call, invoked by a client on an EJB object, is detected by the *Invoker MBean* that provides remote method invocation over a variety of protocols. The Invoker passes the call to the MBean Server, which in turn forwards it to the appropriate service. If the method call is directed to the EJBs of an E-business application hosted by the application server, then the call is passed to the EJB container.

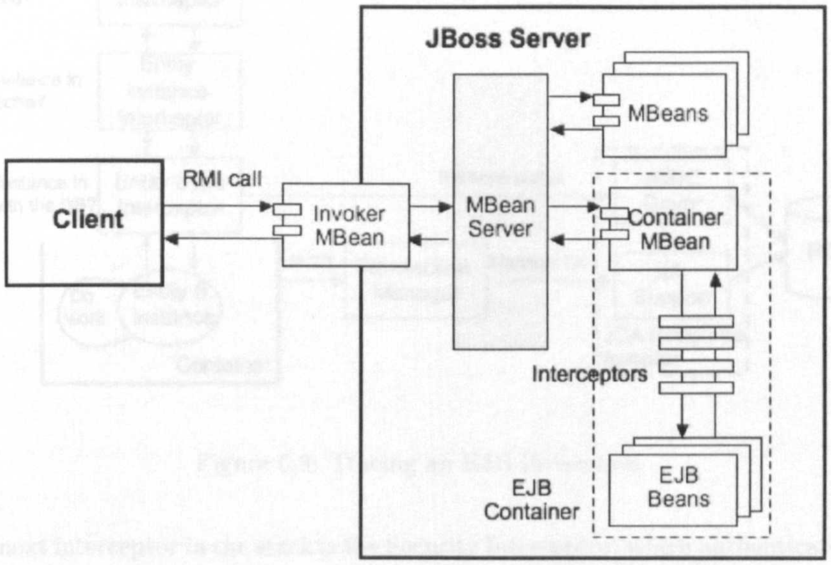


Figure 6.2: Architecture of the JBoss Application Server

Before reaching the EJB, the method call traverses the *Interceptor stack*, a chain of plugin objects interposed across the method invocation path. Interceptors are used to provide EJB services, such as transaction, security and managed persistence, to the method calls performed by the client. Information about the state of the call is kept in the context of the invocation. Each interceptor reads, adds or removes data from this context and performs the related operation or implementation decision, eventually, it explicitly calls the next interceptor in the stack. There is a clear functional distinction between different Interceptors, which allows the deployer of the EJB-application to vary the behavior of the container to a great degree. This also allows new plugins to be deployed, which can bring in new functionalities.

6.3.1 Tracing an EJB Invocation

Figure 6.3 shows a typical method invocation on an Entity Bean traced through the Interceptor stack.

Upon receiving an inbound method call, the Transaction Interceptor decides how to handle this invocation by calling the Transaction Manager service.

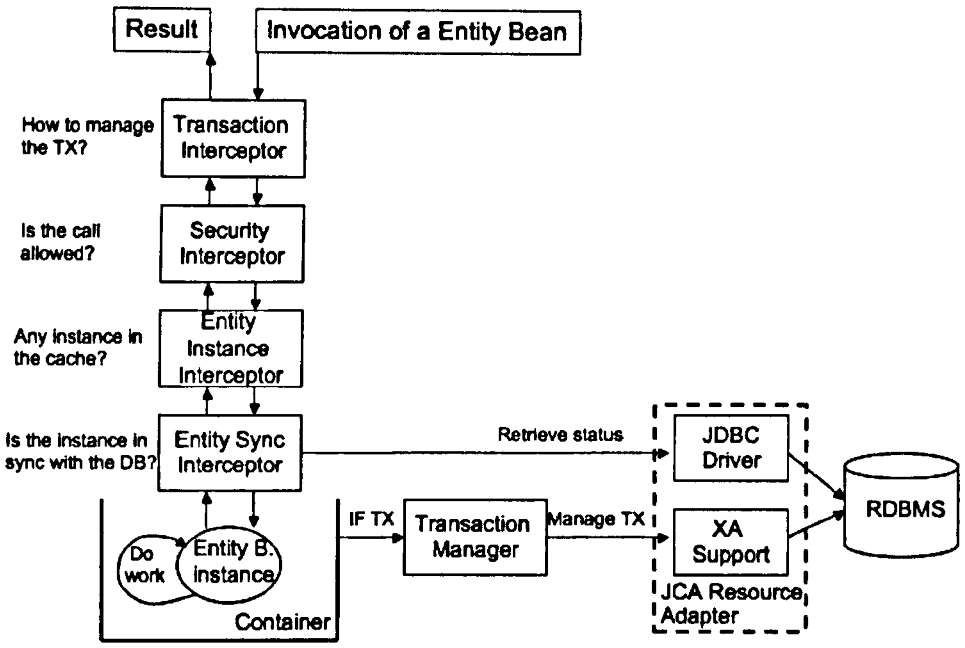


Figure 6.3: Tracing an EJB Invocation

The next interceptor in the stack is the Security Interceptor, which authenticates checks if the caller is allowed to perform that particular method invocation.

In order to invoke a business method on the EJB instance, the instance must be acquired. To do this the Instance Interceptor checks if there is already an instance in the cache. If not, a new one must be created and synchronized with the persistent state. The Entity Synchronization Interceptor performs this operation, loading the state of the bean from the database.

Finally, the container is invoked and the call is delegated to the EJB instance. The instance performs some work, and returns a result. The interceptor stack is now followed in reverse by having each interceptor return from the invoke operation.

If the transaction does not end with this call, the instance becomes locked so that no other transaction may use it for the duration of the current one. The Transaction Interceptor handles the method return according to the transaction settings. Generally, if the transaction is marked for rollback or a system exception is thrown then the overall transaction is rolled back and a *Transaction Rolledback Exception* is thrown, otherwise, the transaction is committed. The call completes with the MBean Invoker returning the result to the client.

6.3.2 Accessing the Database

During its life cycle, the bean performs some database accesses to browse, retrieve or store enterprise data. These are SQL operations at the database, to search, read or write data on the database tables. Data can be retrieved from a relational database, as well as from other non-Java legacy applications, such as enterprise information systems, transactional resource managers, or mainframes.

Some of these accesses can be performed using a transactional access: the database (or the legacy application) checks for conflicts as part of acquiring a write lock, and, if there are conflicts, i.e. read or write locks already acquired by others, then it will not get the write lock. If a lock conflict occurs, the transaction is automatically rolled back and an exception will be thrown to the application. Another possible implementation is to block the thread when a conflict is detected, until the current locks are released, due to commit or roll back of other transactions.

The behavior after a transaction failure depends on the type of implementation: in the CMP the transaction rolls backs immediately; in the BMP the programmer decides whether to return an error or retry.

The Transaction Manager is the service that assumes the responsibility for enabling transactional access to an EJB and maintains the persistent state of the bean involved in the transaction.

Transactional accesses can be *local* or *global*. The local transaction runs purely in the database scope and commits or rolls back within a single database interaction; it is managed internally by the database without the use of any external Transaction Manager. A global transaction (or *XA transaction*) is the one that is controlled within the application server and may encompass other application servers and database. For this reason it is managed outside the database with the help of an external Transaction Manager. A local transaction that is used within a global transaction is subordinated to the global transaction, i.e. it cannot commit or roll back independently.

The persistent state of the bean is maintained in the database. The database is controlled by its own Resource Manager that can be accessed using the *Java Database Connectivity* (JDBC) driver. To enable the Resource Manager to participate in a transaction, the *XA Resource* interface is required. Therefore the Transaction Manager interoperates with the Resource Manager via the XAResource interface, while the application interoperates with the Resource Manager via the JDBC driver. In JBoss both these functionalities are included in the *Resource Adapter* component.

A sample scenario [28] of a single transaction involving three enterprise beans and two resource managers is shown in Figure 6.4. A session bean receives a client invocation

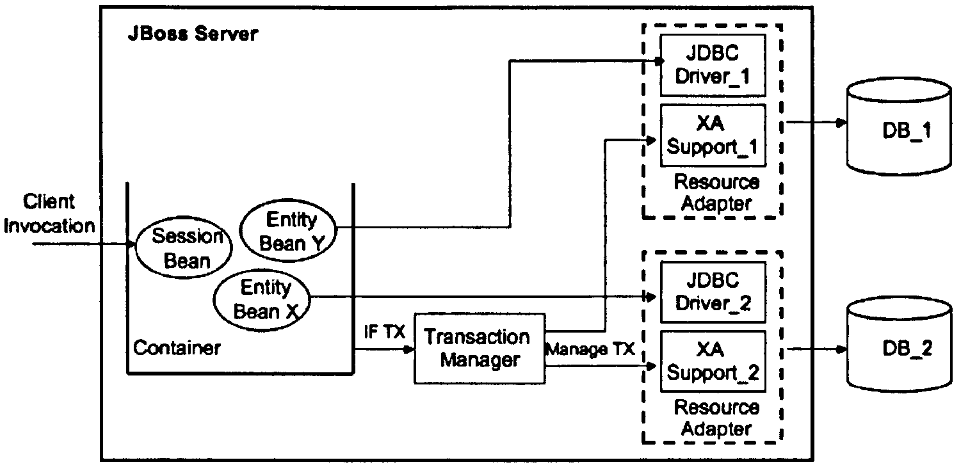


Figure 6.4: Elements Involved in EJB Transactions

initializing a transaction, say T1, involving a number of invocations on two entity beans (X and Y).

When the entity beans are required, the session bean needs to activate these beans via their home interfaces, which results in the retrieval of their states from the appropriate Resource Managers for initialising the variables of X and Y.

The container is responsible for ensuring that the Resource Managers are kept informed of transaction starts and ends. In particular, when the persistent state is retrieved from DB1 (or DB2) at the start of T1, the Resource Manager acquires a write lock for the resource. This prevents other transactions from accessing the resource until T1 ends (commits or rolls back). Moreover the XA resources (XA1 and XA2) register themselves with the Transaction Manager in order to take part in two-phase commit protocol that lets resources agree to commit a transaction.

Once the session bean has indicated that T1 is at an end, the transaction manager attempts to carry out two-phase commit to ensure all participants either commit or rollback T1. Therefore the Transaction Manager polls DB1 and DB2 to ask if they are ready to commit. If any of them cannot commit, they inform the Transaction Manager who tells all the participants to roll back. If the Transaction Manager receives a positive reply from the databases it informs all participants to commit the transaction and the modified states of X and Y are persisted.

The Resource Adapter plays a central role in the integration and connectivity since it mediates between the application server and the database. It serves as point of contact and provides the ability for the application server to manage connection pooling and transaction management for the database. The Resource Adapter provides the *managed* *connections*, which are wrappers for the real JDBC physical connections to the database. Live connections are maintained in pools, ready to be used by client requests in the application server. Once the request is done with it, the connection returns to the pool. If there are more requests than database connections, the requests wait in a queue to get the first connection available.

The reason for providing the pooling functionality is that connections to a relational database take time to set up and consume a significant amount of resources. It would be unacceptably inefficient to construct, use, and tear down a physical connection for each client request in an application server environment, nor would it usually be appropriate to reserve a connection for each user. Some means of sharing and reusing expensive connections is necessary.

6.4 The QoS Control System

The QoS Control System is the infrastructure that we designed to enhance the JBoss application server (V. 3.2.6) with QoS monitor and control functionalities. The main components that form the QoS Control System are illustrated in Figure 6.5, these are:

QoS Control Unit (QoS CU). This component contains the core logic for analyzing QoS values and comparing them with some given tuning policies.

Monitor Subsystem. The monitor component supplies the mechanism for collecting, filtering and reporting performance data.

Actuator Subsystem. The actuator component acts on the tuning knobs to vary the configuration of the controllable parameters.

These components are akin to the brain, the eyes and the hands of a human working on a *Target System.* The Target System represents the base system being managed, in our case the JBoss application server.

6.4.1 QoS Controller Unit

The QoS-CU analyses the actual QoS values, obtained from the Monitor Subsystem, in order to asses compliance with some tuning policies. If policies are not met, the controller should determine what actions to take.

Our analytical model is designed to serve as core logic for the QoS-CU, being its mathematical foundation for estimating the tuning policies. In Chapter 7 we employ the QoS Control System to quantify the static parameters of the model and to carry out

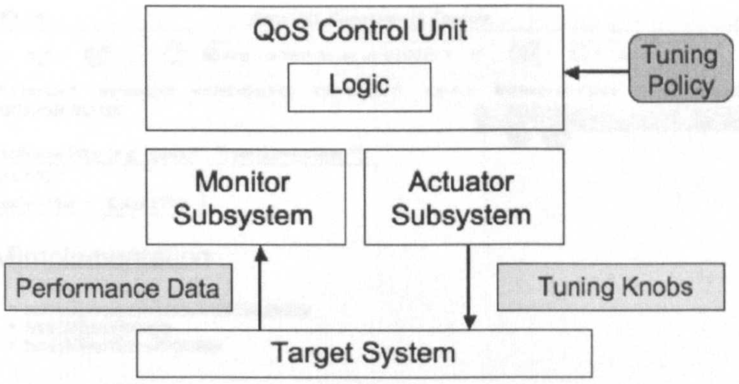


Figure 6.5: Components of the QoS Control System

the configuration tuning. We show that, after appropriate calibration tests, the model is effective in predicting the performance of the target system under given load conditions and in choosing the optimal settings for the controllable parameters with respect to specified performance measures.

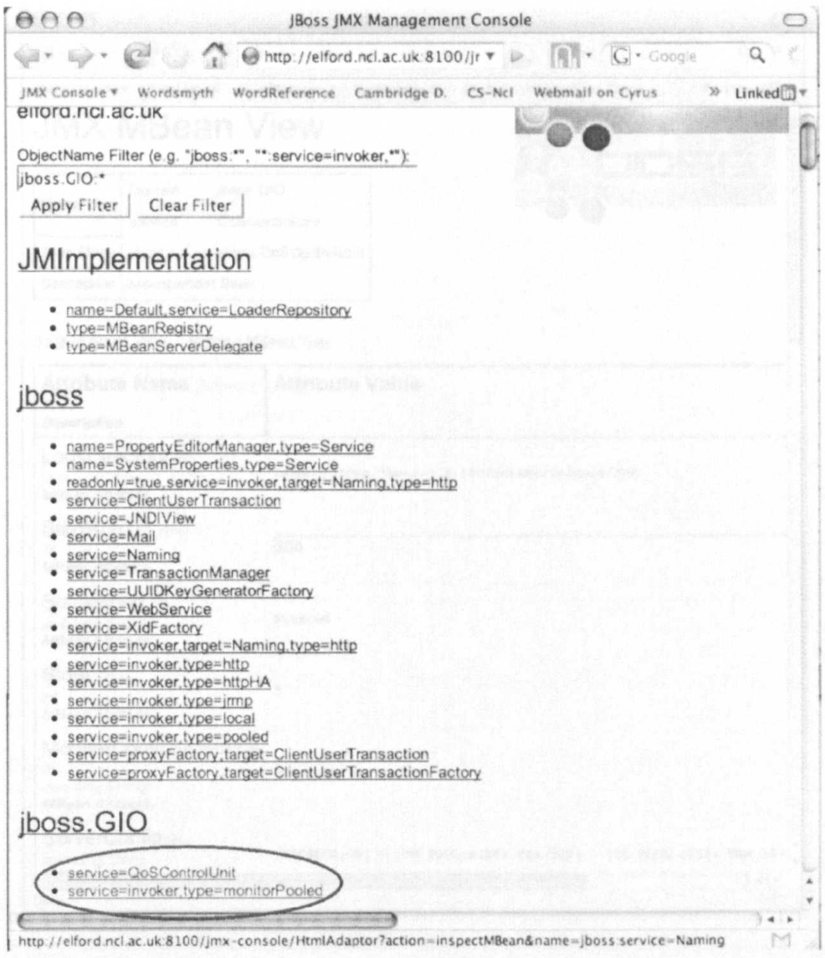
The actual implementation of this process is human driven, since the QoS-CU offers an interface to interact with an external user, who runs calibration tests, deems models estimations, and provides the tuning policies. For instance, the new server configuration can be selected by the external user among a set of satisfactory configurations found observing the graphs that plot model estimations. However, extending it to an automated process, to obtain a full automated tuning system is a smooth operation, as we explain further in our future works, Chapter 8.

The QoS-CU is implemented as an MBean and is registered with the MBean server as any other JBoss service, Figure 6.6 illustrates the JMX Agent View of the enhanced JBoss with our QoS services.

When the application server is running, the QoS-CU MBean exposes its attributes and management operation on the interface illustrated in Figure 6.7. Exposed attributes are, for instance, the Actual QoS achieved by the application server; the position within the file system of the *logs directory*, the repository for storing performance data reports; the status of the Monitor and of the Controller Subystem. Among the several management operations, there are the methods to start and stop the Monitor and Controller Subsystem, to update the data on node QoS, to start the Testing Phase used for the calibration process.

Both the Monitor and the Actuator Subsystems are started and managed by the QoS-CU MBean. At application server startup, the MBean server starts the QoS-CU MBean, which in turns starts the two subsystems.

It is worth observing that our implementation exploits JBoss extensibility, therefore



elford.ncl.ac.uk

ObjectName Filter (e.g. "jboss:*", "*:service=invoker,*"):

jboss.GIO:*

JMImplementation

- name=Default,service=LoaderRepository
- type=MBeanRegistry
- type=MBeanServerDelegate

jboss

- name=PropertyEditorManager,type=Service
- name=SystemProperties,type=Service
- readonly=true,service=invoker,target=Naming,type=http
- service=ClientUserTransaction
- service=JNDIView
- service=Mail
- service=Naming
- service=TransactionManager
- service=UUIDKeyGeneratorFactory
- service=WebService
- service=XidFactory
- service=invoker,target=Naming,type=http
- service=invoker,type=http
- service=invoker,type=httpHA
- service=invoker,type=jrmp
- service=invoker,type=local
- service=invoker,type=pooled
- service=proxyFactory,target=ClientUserTransaction
- service=proxyFactory,target=ClientUserTransactionFactory

jboss.GIO

- service=QoSControlUnit
- service=invoker,type=monitorPooled

http://elford.ncl.ac.uk:8100/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss:service=Naming

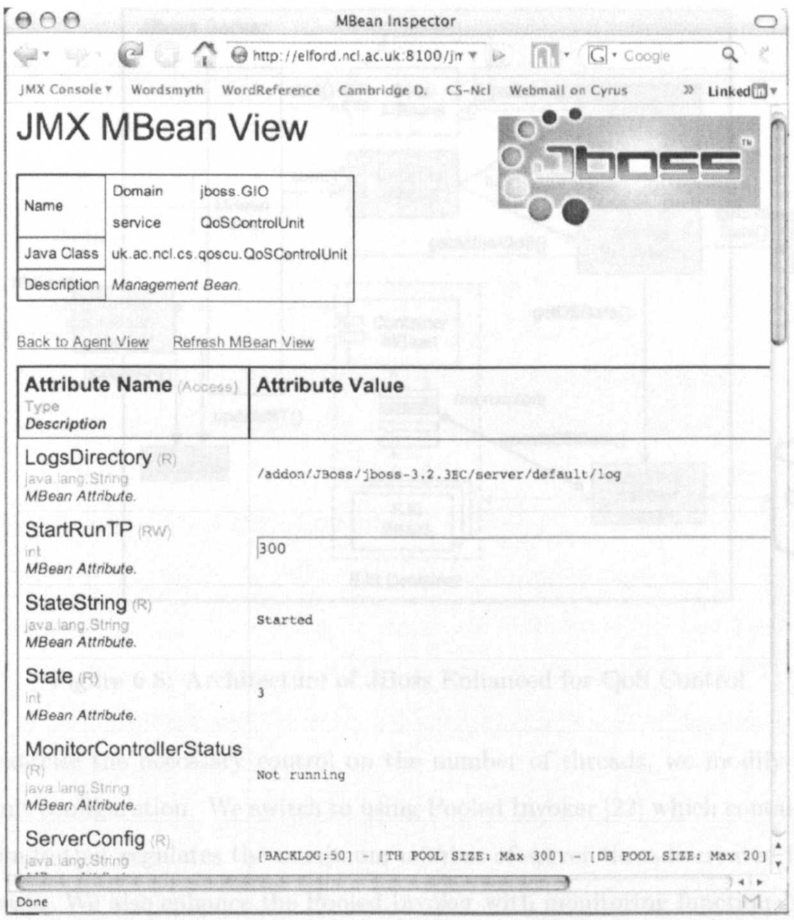
Figure 6.6: Enhanced JBoss: JMX Agent View with QoS-CU

the new components are dynamically deployed on the application server and are fully integrated with it. The JBoss architecture, previously depicted in Figure 6.2, is now enhanced with a set of new components, as it is illustrated in Figure 6.8, where the shaded objects correspond to our QoS Control System components. A detailed description of the single objects is provided in the following sections.

6.4.2 Monitor Subsystem

The Monitor Subsystem is made of a number of monitoring components, or *sensors*, which collect data samples from the application server. These samples are filtered and correlated in order to obtain the key static parameters used in the calibration of the model, as will be described in Chapter 7.

The Monitor Subsystem periodically polls the sensors and records the data in the


MBean Inspector

http://elford.ncl.ac.uk:8100/jn

JMX Console Wordsmyth WordReference Cambridge D. CS-Ncl Webmail on Cyrus Linked

JMX MBean View

Name	Domain	jboss.GIO
	service	QoSControlUnit
Java Class	uk.ac.ncl.cs.qoscu.QoSControlUnit	
Description	*Management Bean.*	

Back to Agent View Refresh MBean View

Attribute Name (Access) Type ***Description***	**Attribute Value**
LogsDirectory (R) java.lang.String *MBean Attribute.*	/addon/JBoss/jboss-3.2.3EC/server/default/log
StartRunTP (RW) int *MBean Attribute.*	300
StateString (R) java.lang.String *MBean Attribute.*	Started
State (R) int *MBean Attribute.*	3
MonitorControllerStatus (R) java.lang.String *MBean Attribute.*	Not running
ServerConfig (R) java.lang.String	[BACKLOG:50] - [TH POOL SIZE: Max 300] - [DB POOL SIZE: Max 20]

Done

Figure 6.7: Enhanced JBoss: View of the QoS-CU MBean

Actual QoS object, which is made available as an attribute of a JMX MBean. The data are permanently persisted in report files, saved in the application server file system for further reuse. The length of the polling interval can be set using the QoS-CU MBean interface.

The following sensors are used by the Monitor Subsystem:

Invoker MBean Sensor. Incoming client invocations are passed on to the application server by the Invoker MBean. By default, JBoss uses the JRMP Invoker MBean, an RMI implementation that, when used as the Invoker in a remote client, allows invocations from the RMI/JRMP protocol into the JMX MBean Server.

More precisely, the JRMP Invoker passes remote client invocations to the server threads running on the application server process, thereby creating a new thread for every RMI request that comes in. This does not allow control on the number of server threads concurrently running on the application server.

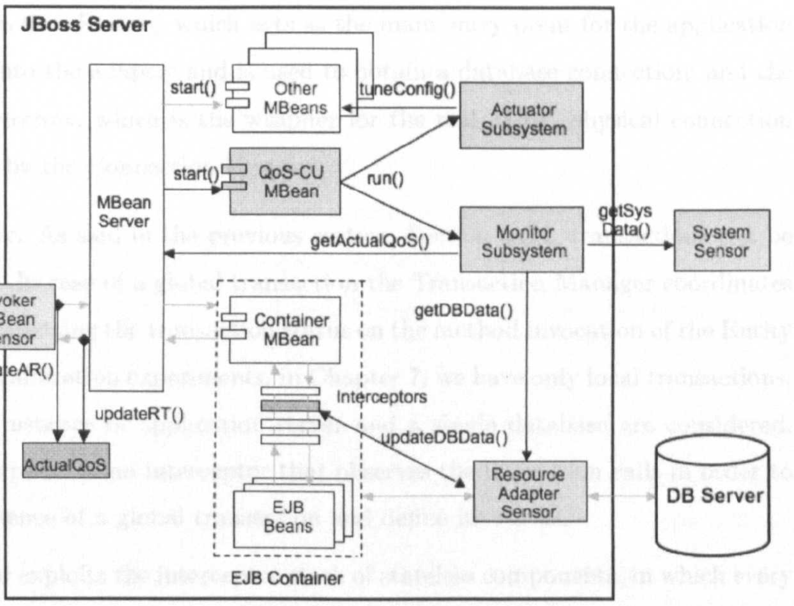


Figure 6.8: Architecture of JBoss Enhanced for QoS Control

To exercise the necessary control on the number of threads, we modify the JBoss default configuration. We switch to using Pooled Invoker [22] which contains the key attribute that regulates the maximum number of server threads created for serving requests. We also enhance the Pooled Invoker with monitoring functionalities. This enhancement allows us to observe inbound and outbound invocations (needed to compute *Arrival Rate* and *Throughput*) and the time taken by the server to process an invocation (to compute *Response Time*). These performance data are collected in the ActualQoS object which is made available by the MBean Invoker to the other components through the MBean Server. The Monitor Subsystem retrieves the *ActualQoS* invoking the `getActualQoS()` method on the MBean Invoker interface.

Resource Adapter Sensor. As stated in Chapter 3, a request at the application server may involve a random number of accesses to the database. These accesses are either lightweight accesses, for instance a read-only query for retrieving the bean state, or transactions that modifies enterprise data, which typically takes longer and may terminate successfully (commit) or unsuccessfully (roll back).

In order to collect data on transactional and non-transactional jobs, we implement a wrapper object for the JCA Resource Adapter and deploy it in the application server. Our Monitor Resource Adapter contains functionalities that allow tracking the performance of database connections.

The JBoss objects that are modified and instrumented to collect data on method calls are: the *Connection Factory*, which acts as the main entry point for the application server to call into the adapter and is used to obtain a database connection; and the *Managed Connection*, which is the wrapper for the real JDBC physical connection and is created by the Connection Factory.

Interceptor Sensor. As said in the previous section, Section 6.3.2, transactions can be local or global. In case of a global transaction the Transaction Manager coordinates the transaction, setting the transaction status on the method invocation of the Entity Beans. In our calibration experiments, in Chapter 7, we have only local transactions, since a single instance of application server and a single database are considered. However we implement an interceptor that observes the invocation calls in order to detect the presence of a global transaction and define its status.

The interceptor exploits the interceptor stack of stateless components, in which every call proceeds through the stack from first to last, until the target entity bean is called. After the entity bean has finished with its method, the call will unwind through the stack in reverse order, and the interceptor detects the status of the transaction, such as active, committed, rollback etc.

Moreover the Interceptor Sensor traces rollback and timeout exceptions thrown by the application, respectively caused by transactions that are rolled back or by a timeout occurring while waiting for a Managed Connection to become available.

The collected data are sent to the Resource Adapter Sensor which computes the service time at the database and rate of transactional requests.

System Sensor. The System Sensor collects the operating system statistics. To get data on memory usage we instantiate the Runtime Java class and use the method call `getRuntime()`. However this class returns only data related to the JVM in which the application server is running.

To collect data on CPU usage we have to move beyond this point and reach the machine system calls using Java Native Interface (JNI).

In order to implement the JNI system calls, the System Sensor object is implemented as a wrapper class of a *Dynamic Link Library (DLL)* that performs low-level method calls on the Linux operating system functions (the application server runs on Linux Fedora Core, release 4).

The class acts as an interface and loads the DDL library when it is called for the first time. The DDL library calls the standard public functions in the extension DLL to obtain the operating system statistics through JNI.

The CPU usage is retrieved from the operating system function `getrusage()`, which returns the number of milliseconds of CPU time is used by a process with the given PID. Since Linux does not keep a table of the CPU time used by a single children process (unless this process is terminated and its parent has waited for its signal) then it is not possible to compute the CPU usage for every active thread. We consider instead the CPU usage of the entire application server process, as further explained in Section 7.3.1.3. Data on CPU and Memory usage are then periodically collected by the Monitoring Subsystem.

For collecting data samples we pursue a "poll approach". This means that sensors actively collect data, then the Monitor Subsystem periodically polls sensors for the data. Data are transfered in form of serializable objects by implementing the `java.io.Serializable` interface, and letting Java handle the serialization internally. The advantages of serialization are robust persistence and the efficient distribution. The drawback is that the process of marshalling/demarshalling presents some overhead, but this this can be avoided by making sparse polling intervals or reducing the serialized object size.

An alternative could have been a "push approach", where sensors use messages to sent data updates at the central Monitor Subsystem, for instance by the Java Message Service (JMS) architecture [57]. According to Sun's specifications, it enables distributed communication that is loosely coupled and asynchronous. Only JBossMQ[21], the JMS version implemented on the JBoss application server, is not reliable in collecting performance metrics. During the experiments some messages containing data samples can be lost, altering the final resuslts. The JBoss developers are currently working on a more efficient solution, JBoss Messaging [20] that will replace JBossMQ in JBoss application server 5.0 (not available as of the time of writing).

6.4.3 Actuator Subsystem

The Actuator Subsystem has the duty of performing the API calls on the other MBeans in order to adjust the controllable parameters. These parameters are dynamically controllable and can be tuned at run-time. The same parameters are considered in our simulation and calibration experiments, where we observe the behavior of the Target System at the variation of the system configuration.

The controllable parameters are:

Backlog. This parameter represents the maximum queue length for incoming request to connect sent by an external client. If the request arrives when the queue is full, the connection is refused. In the JBoss application server the default for this parameter is 200.

Application Server Thread Pool Size. This parameter represents the size of the pool containing the server threads that process client requests. The parameter controls the maximum number of server thread that can be concurrently running, and thus the number of requests that can be concurrently executed. In JBoss the default value is 300.

Database Connection Pool Size. This parameter represents the size of the pool in which are maintained live connections to the database. The parameter controls the number of connections that can be concurrently handled by the application server. In JBoss the default value is 20.

Database Queue Timeout. This parameter represents the maximum time to block while waiting for a database connection to become available before throwing an exception. In JBoss the default value for this parameter is 5000.

The server parameters are exposed as management attributes on the QoS-CU MBean interface, along with their actual set up. The Actuator modifies the parameters as result of the tuning policies provided by an external user. In our specific case this user is the one who runs the calibration experiments. Besides, the parameters could also be modified according to some internal policies, for instance if the optimal configuration setup were automatically determined.

6.5 Collected Data and Controllable Parameters Within the Target System

In Chapter 7 the calibration process is carried out employing the QoS-CU to evaluate the key parameters of the model and to compare the observed performance with those predicted by the model.

The key parameters are obtained by data samples gathered from the different sensors. The sensor position within the Target System is illustrated in Figure 6.9, where numbers correspond to the key parameters listed in the following list.

1. *Total CPU service time*, $[st_{CPU}]$: the observed CPU service time, monitored by the CPU sensor.

2. *Average Memory usage*, $[u_{Mem}]$: the memory usage, monitored by the CPU sensor.

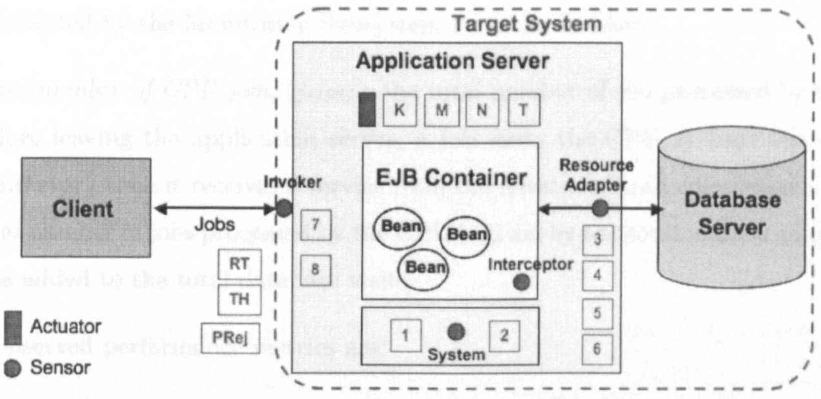


Figure 6.9: Sensors Position Within the Target System

3. *Transactional job/Non-transactional job service time*, $[\xi_T, \xi_{NT}]$: The database service times, collected by the entity beans interceptor and the sensor on the database resource adapter.

4. *Probability of leaving the system from CPU* (instead of going to database), $[\beta]$: estimated as:

$$\beta = \frac{1}{avgV} ; \tag{6.1}$$

where $avgV$ is the average number of visits that each job makes to the database, given by the ratio of total number of database visits (v_{DB}) to total number of jobs leaving the system (j):

$$d = \frac{v_{DB}}{j} ; \tag{6.2}$$

5. *Probability of being a transactional job* (instead of NT), $[\theta]$: transactional jobs are those that terminate with commit or roll back, collected by the sensor on the database resource adapter. θ is estimated as the ratio of total transactions (tx) to total number of database visits:

$$\theta = \frac{tx}{v_{DB}} ; \tag{6.3}$$

6. *Probability of having a committed transaction* (instead of rolled back transaction), $[\gamma]$: rate of rolled back transactions, collected by the entity bean interceptor checking the error code; the same sensor collects data on timed out jobs. γ is estimated as the ratio of total number of committed transactions (c) to total transactions (tx):

$$\gamma = \frac{c}{tx} ; \tag{6.4}$$

7. *Jobs arrival rate*, $[\lambda]$: The rate at which job requests from clients arrive. The data

is collected by the Monitoring Subsystem from the Invoker.

8. *Total number of CPU jobs*, [j_{CPU}]: the total number of job processed by the CPU. Before leaving the application server, a job visits the CPU at least one time and again every time it receives a service from the database, until completion. Then the total number of jobs processed by the CPU is given by the total number of completed jobs added to the total database visits.

The observed performance metrics are:

RT: *response time*, which is the time taken by the application server to process an accepted job;

SuccRT: *successful response time*, which is the time taken to "successfully" process an accepted job;

TH: *throughput*, which is the average number of jobs processed by the application server per unit of time;

SuccTH: *successful throughput*, which is the average number of jobs that successfully complete per unit of time;

PRej: *probability of rejection* The ratio of jobs that are unsuccessfully processed (un), due to rejection at the application server queue, database requests that have timed out or to transactions that have rolled back, to total number of jobs leaving the system (j):

$$PRej = \frac{un}{j}\ ; \qquad (6.5)$$

The term "successful" refers to the fact that the job is not rejected at the application server queue, nor at the database queue and the related transaction commits. For the purpose of the calibration process, this term is eventually redefined in Section 7.3.1.

Figure 6.9 also shows the controllable parameters that can be tuned by the Actuator:

K: the Backlog parameter;

M: the Thread Pool Size parameter;

N: the Database Pool Size parameter;

T: the Database Queue Timeout parameter.

During the calibration process, the Actuator provides support in tuning the configuration set-up, so that the effect of the different parameters on the performance can be examined.

6.6 Summary

In this chapter we illustrate the implementation of a QoS Control System that enhances with QoS control functionalities a popular open-source application server, JBoss. The system is fully integrated in the JBoss application server platform and is capable of monitoring application server performance, as well as modifying its configuration parameters towards some QoS objectives.

Firstly, we provide an overview of the main technologies employed, describing the internal structure and the interactions within the Target System. This can be also considered as an evidence of the concepts defined in Chapter 3, where the system model is analyzed.

Then we discuss the implementation details of the three components that form the QoS Control System: (a) the QoS-CU MBean, which determines the configuration parameters need to be tuned and to what values; (b) the Monitor Subsystem, composed by several sensors which collect data samples from the application server; and (c) the Actuator Subsystem, which acts on the tuning knobs to vary the configuration of the controllable parameters.

The QoS Control System will be extensively used in the following chapter (Chapter 7) during the calibration process for observing a real E-business system over a range of several system configurations. Using the QoS Control System, we evaluate the key parameters of the model and collect data on observed performance. The key parameters are then substituted in the approximate solution and the estimated performance is compared to the observed one. This will demonstrate that our approximate solution can predict the performance of the real system and assess the optimal system configuration very accurately.

The QoS Control System can be then used to monitor the application server, to tune the configuration setup and to accomplish the testing phase for the model calibration process. To the best of our knowledge this is the only implementation of QoS Control System for the JBoss application server that, used in combination with the approximate solution, can estimate the performance and predict the optimal configuration setup.

Chapter 7

Validation of Approximation through Calibration

7.1 Introduction

This chapter illustrates the calibration and validation of the analytical approximation that is presented in Chapter 4. Model calibration is the process of estimating the static parameters of a model to obtain a match between observed and estimated behaviour [46].

Tests are run in order to observe the behaviour of an E-business system of realistic size and complexity. The static parameters are obtained by data samples collected using the QoS Control System, described in Chapter 6. These parameters are substituted in the analytical model and the estimated performance is compared with the observed one in order to assess the correspondence between our model and the real system.

The E-business system examined here comprises the ECperf application deployed on the JBoss application server, connected to the Oracle database server. However the analytical approximation and the calibration process are not tied to a specific technology. The same methodology could be applied to other application servers or E-business applications.

In this chapter we first describe the Target System and the E-business application that we consider and the methodology that we follow in our tests. We also describe the problems that need to be tackled in calibrating the model. We finally present the experimental results that show a close match between predictions and observations. The graphs demonstrate that our approximation solution is effective in estimating the performance of a real system, and also it is precise in predicting the optimal configuration setup.

7.2 Target System

The E-business system that we consider for the observations is constituted by a J2EE application, the *ECperf* benchmarking application [59], hosted by a 2-Tier system. This includes a single application server node, the JBoss application server, and a database, the Oracle database server, shared among other unknown users. We refer to it as the *Target System*.

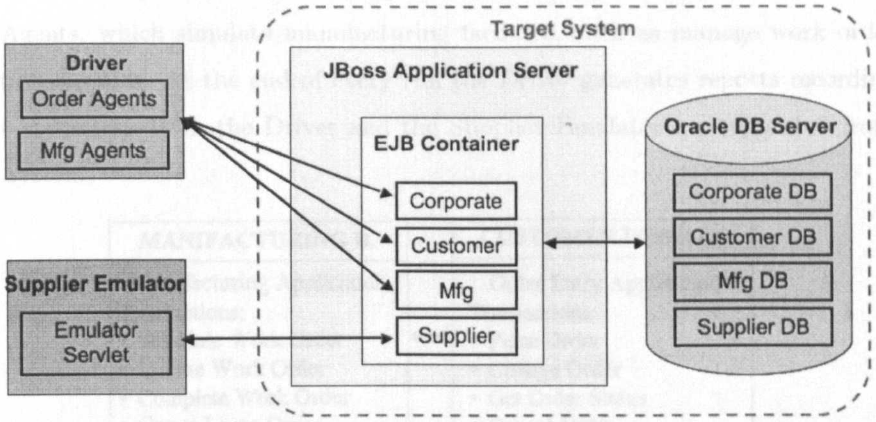


Figure 7.1: Target System with ECperf Environment

ECperf is the benchmark for measuring performance and scalability of E-business systems built by Sun in conjunction with J2EE application server vendors. It is a publicly available industry-standard benchmark, built and maintained by the industry itself. Besides, it has a particular attraction of not having been tailored to any specific vendor. The application design reflects the state of the art in the design and implementation of distributed applications [9]. For this reason it was taken up by the Standard Performance Evaluation Corporation (SPEC) to develop the *SPECjAppServer* [52] for the standardized suite of benchmarking applications with application server focus.

ECperf has been mainly chosen because it has the characteristics of realistic enterprise application for a global corporation. The series of simulated events that represent the the business problem modelled are based on manufacturing, supply chain management and order/inventory, all of them requiring the use of many middleware services, such as distributed transactions, naming services, fault-tolerance, caching, object persistence and resource pooling. These services of application servers are specifically exercised by the application workload.

Note that ECperf is used as an illustrative complex E-business application that drives load on the application server, rather than a real benchmarking application that assess the performance. For this reason we will omit to mention ECperf reports in our discussion.

Figure 7.1 illustrates the Target System and its ECperf environment. The workload is generated by the ECperf Supplier Emulator and the Driver. The former is implemented as a Java servlet in a separate web container and simulates the interactions of the system with suppliers sending and receiving purchase orders to/from suppliers. The Driver spawns a variable number of threads that represent clients sending requests to the Target System. Two types of threads are generated: the Order Agents, which simulate customer operations such as place orders, make changes, check the order status; and the Manufacturing Agents, which simulate manufacturing facilities, such as manage work orders or production outputs. At the end of every run the Driver generates reports recording the relevant statistics. Both the Driver and the Supplier Emulator are separated from the Target System.

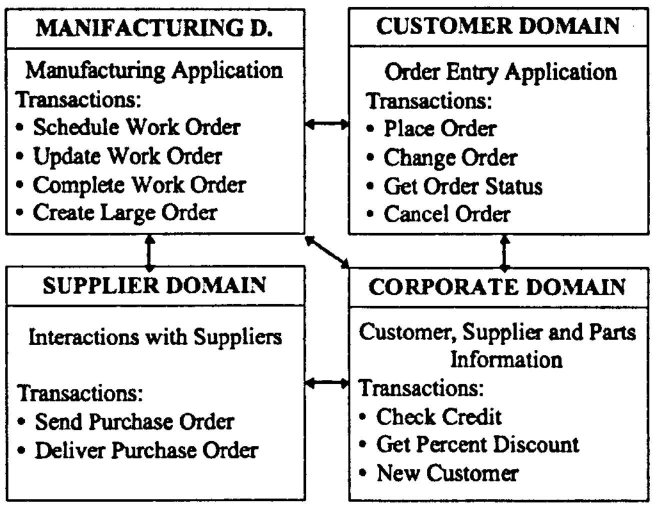


Figure 7.2: ECperf Domains and Main Transactions

The core of the application is represented by the EJBs deployed on the application server container. These EJBs implement the logical entities that describe the distinct business sphere of operations. They reproduce the four domains of the business model with their own transactions: the Customer Domain, handling customer orders and interactions; the Manufacturing Domain, performing manufacturing of widgets and production line operations; the Supplier Domain, handling interactions with external suppliers; the Corporate Domain, managing the data of customers, products and suppliers. Figure 7.2 presents the different domains with their main transactions. All data access operations use entity beans which are mapped to tables in the database. Both container and bean managed persistence is supported.

A relational database is used for data persistence. In order to conform the ECperf specifications we have replaced Hypersonic, the JBoss default database, with the Oracle database because the former does not support distributed transactions, which are fundamental in ECperf operations.

The Driver sends requests to the Target System in the form of units of work, which initiate the business transactions involving one or more remote method calls to the EJBs. Each high level action, such as customer making a new order or a manufacturer updating the status of an existing order, is defined as a business operation. The arrival rate of requests is directly related to the chosen *Injection Rate* (IR) that refers to the rate at which business transaction requests from the Driver are injected into the Target System. An IR increase causes the arrival rate increase. To stress the ability of the application server to handle concurrent sessions, the ECperf requires a minimum number of Order Agent threads equal to $5 * IR$. The number remains constant over the course of a benchmark run. The Manufacturing Agent scales in a similar manner, the number of threads in this case is equal to $3 * IR$. For instance, if $IR = 30$, the number of Order Agent threads is $5 * IR = 150$ and $3 * IR = 90$ is the number of Manufacturing Agent threads. The result is a total of $150 + 90 = 240$ simulated customers concurrently running and sending requests to the application server.

7.3 Calibration and Testing Phase

We define *Testing Phase* as the process of running the experiment and collecting data samples at the end of the run. In order to prove our concepts we run a series of experiments tuning the system configuration. As mentioned in Chapter 3, the application server is provided with tunable "knobs" or configuration parameters that can be dynamically adjusted at server run-time to optimize the performance. For our experiments we take into consideration the following parameters:

M: the *Thread Pool Size* parameter which controls the maximum number of application server threads concurrently running;

N: the *Database Pool Size* parameter which controls the maximum number of open database connections. These connections are the physical entities representing the database servers in the analytical model;

T: the *Database Queue Timeout* parameter which sets the maximum time to block while waiting for a database connection to become available before throwing an exception.

At the beginning of each run, we set the configuration to certain values and start the process along with the ECperf Driver. All this can be done connecting to the QoS-CU

MBean interface which lists the Start Testing Phase method among the several management operations, as described in Section 6.4.1.

At Testing Phase the QoS-CU gathers data samples that define the key parameters of the model. Data samples are collected when the system is at steady state. At the end of the steady state, a report file (*QoS.data*) is saved in the application server *log* directory. Length of the Testing Phase, length of the Steady State interval, position of the report file within the file systems are all attributes exposed on the QoS-CU MBean interface.

The data used to derive the key parameters are collected by means of the Monitor Subsystem using several sensors within the Target System, as it is described in Chapter 6.

For the purpose of calibrating the model, those key parameters are substituted in the analytical approximation and estimates of the performance metrics are obtained. Estimates are then compared with the performance metrics observed at Testing Phase, the validation of the accuracy is discussed in section 7.4. The list of the data samples collected by the sensors and the performance metrics considered for comparison are specified in Section 6.5.

7.3.1 Problems Tackled in Calibrating the Model

During the calibration process, several problems need to be tackled. These problems mainly raise because both the benchmarking application and application server are open source applications. Sometimes the problem are due to poor code writing (like we would consider in JBoss the Timeout Effect illustrated in Section 7.3.1.4), sometimes due to the nature of the application (like the ECPerf case discussed in Section 7.3.1.1). This section lists the problems and how they have been dealt with.

It is worth noting that the differences between the analytical model and the real system are addressed with workarounds that exclude the alteration of the analytical model. We have made this choice so that the analytical model remains general, rather than reviewing our model to closely match the specific technology used for our experiments

7.3.1.1 Application Server External Queue

ECperf, being a tool for benchmarking, is designed to test the Target System performance under heavy workload. In testing, it excludes from the Target System the network and the external connections, behaving as if there is an infinite queue of jobs. Whereas our analytical model more realistically regards the application server with an external queue, where connection requests queue until a server thread becomes available.

Furthermore, we have observed that when an ECperf emulated client obtains a connection with a server thread, it holds it until the end of run sending a continuous stream of jobs. If the number of clients is bigger than the number of server threads, the discarded clients are ignored. Otherwise, if the number is smaller, some server threads are idle and excluded from the computation. In either case the external queue becomes meaningless. This differs from our model, where a new job entering the system is taken directly from the external queue by the first server thread available; at the end of the computation the job leaves the system and the thread is freed.

Adapting the activities of the benchmarking application to our model could alter ECperf functions. Moreover, network congestion problems can affect the rejection as well as application server congestion, therefore an effective evaluation of the probability of rejection could be done only at the client side, there where the error to connect becomes evident. But this approach is invasive of the client space, since it implies the development of a sensor at the client side or at the JBoss client proxy, in either cases the sensor is installed in the client address space.

However we observe that at the steady state, when all the threads are active, there is a continuos stream of jobs sent by each client. Besides these jobs are sent in a synchronous manner, directly connecting to the server threads. This is a close approximation of the infinite pool of jobs case of our analytical model. Therefore we decide to only take into account the analytical approximation case with $PoolofJobs = \infty$ and exclude the external queue from the observation of performance values.

The result is that there is no rejection at the application server external queue.

7.3.1.2 Server Thread Overhead

Although the server thread is busy serving the stream of jobs coming from the same client, it experiences some idle time while terminating the process of a job and starting a new one. This is due to client think time, which is made relevant because of the synchronous nature of the client-server connection in ECperf benchmark. The server threads are always observed as being active by the monitor subsystem, and the idle interval is negligible for a small number of server threads. On the other hand, if we consider a larger number of threads, the overhead accumulates, in fact the idle intervals become evident when there are more than 200 server threads running. As consequence, the overhead affects the model estimates of the performance, as it is shown in Figure 7.13.

Our solution to this is to register the thread overhead and include the observed value to the model estimates. Therefore we replace the Equation 4.15 that computes the response time in Chapter 4

$$W = \frac{L}{T} ; \tag{7.1}$$

with

$$W = \frac{L - o}{T} ; \tag{7.2}$$

where o is the ratio between the total overhead and the observation period.

7.3.1.3 Observability of CPU and Disk

The computation subsystem shown in Chapter 4 is made of the physical resources CPU and Disk. While specific sensors for CPU and Disk monitoring are likely to be available on larger commercial application servers, the open source JBoss application server does not expose the parameters related to these resources, they can be retrieved only monitoring the operating system of the node hosting the application server. The monitoring requires an intrusive sensor into the kernel in order to detect the jobs routing among these resources and the service time the job receives cycling among the resources.

For our experiments we have used a server node running the Linux OS, then we have implemented our own sensor. The application server itself is one of the hundreds of processes running on the kernel, it has the same PID as the JVM and it does not indicate the children processes, which could be the server threads or the single jobs circulating in the system.

This makes difficult the task of collecting detailed samples on service time and number of cycles per job and for this reason we have considered a further approximation for the computation subsystem, as it is described in Section 4.3.1. As we observe with the validation process illustrated in the following section, the approximation results are accurate even with this modification.

7.3.1.4 Timeout Effect

As per our model, a job waits in the database queue to obtain a connection; if its timeout expires, the job is rejected and leaves the system. Instead, JBoss developers have implemented a retry policy at the database connection queue, by which timed out jobs are not directly rejected, but reinserted back in the queue. As illustrated in Figure 7.3, when the timeout expires the job should leave the system. Yet the specific JBoss implementation allows the job to re-enter the queue for three times rearranging its order. This has been done for efficiency, in order to discard a smaller number of jobs, however an adequate tuning of the timeout value produces the same effect. This behaviour is specific to JBoss and differs from our model. It has two side effects: it reduces the number of rejected jobs and it reshuffles the database queue.

If reality were to reflect our model then the number of reinserted jobs would ideally be

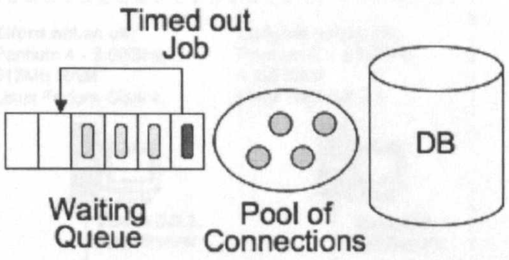


Figure 7.3: Timeout Effect on the Database Queue

zero. The reason for this is that the server thread puts the timed out job straight back into the queue, making the queue always full (in the long term this would overload the database and let more jobs time out). While in our model the timed out job disappears, the server thread gets free and admits a new job for computation, in the meanwhile a new job has the time to join the database queue and being served.

However, we have observed that the default Timeout parameter, $T = 5000ms$ in JBoss 3.2.3, is large enough that only few jobs timeout and rejoin the queue. Besides, a higher setting ($T = 30000ms$) is also recommended by the same JBoss developers [19]. The ratio of timed out jobs is approximately 8% of the overall database accesses, as it is shown in Figure 7.12. This means that a small number of jobs is rejected at the database queue, even with a heavy workload. Therefore the distortion can be deemed negligible.

Finally, after the considerations on the Timeout Effect and the application server external queue, the term "successful job" is re-defined as the job which related transaction commits, while rejection at the application server and database queues is omitted.

7.4 Experimental Results

In this section we compare the performance metrics estimated by the model with those observed on the real system with the purpose of demonstrate the accuracy of our analytical model.

Figure 7.4 shows the environment set up employed for our experiments. The application server used is JBoss 3.2.3, deployed on a Pentium 4 3.00 GHz PC with 512MB of RAM running Linux Fedora Core release 4. The database server used is Oracle 9i (actual version 9.2.0.1.0), deployed on a Pentium 3 931 MHz PC with 1 GB of RAM running RedHat 7.3. The ECperf client is deployed on a Pentium 4 2.80 GHz PC with 1GB of RAM running Fedora Core release 5. The LAN used for the experiments is a 100 Mbps Ethernet.

It is worth noticing that our analytical approximation and the calibration process are

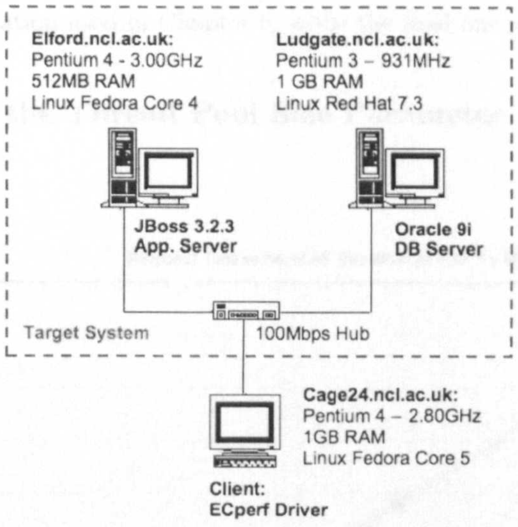


Figure 7.4: Experimental Environment

not tied to a specific technology. The same methodology could be applied to any other application server. The only requirements are: (a) the presence of server knobs that can be used for tuning the system settings and (b) support for a monitor component capable of evaluating the static parameters of the model.

For the scope of this thesis we demonstrate the validity of our approach employing JBoss, however other application servers could be used, such as IBM Websphere [16], Oracle OC4J [47] and BEA Weblogic [4], just to quote the most well known. The same applies to the workload application, here we employ ECperf but Trade 2 from IBM [15], or Sun's Java Pet Store application [58] are equally effective.

The experiments are conducted running ECperf with Injection Rate $IR = 30$, which translates into 120 ECperf clients, as explained in Section 7.2. We observe the behaviour of the Target System under different system configurations, specifically tuning the number of application server threads (M) and the number of open database connections (N). The Timeout (T) variable is set to the JBoss default value $T = 5000ms$ throughout the experiments.

There are four sets of experiments that are organized as follows. The first two sets show the behaviour of the Target System at the variation of M. The purpose of these experiments is to quantify the effect of number of application server thread on the performance of the system. In the first set, the system configuration is analogous to the experiments that validate the approximation through simulation, in Chapter 5. In the second set, the system configuration is adjusted to higher values, similar to those used by default in JBoss. The last two sets of experiments show N varying in a similar manner. The third set reflects the system configuration used in Chapter 5, while the final one presents higher settings.

7.4.1 Tuning the Thread Pool Size Parameter

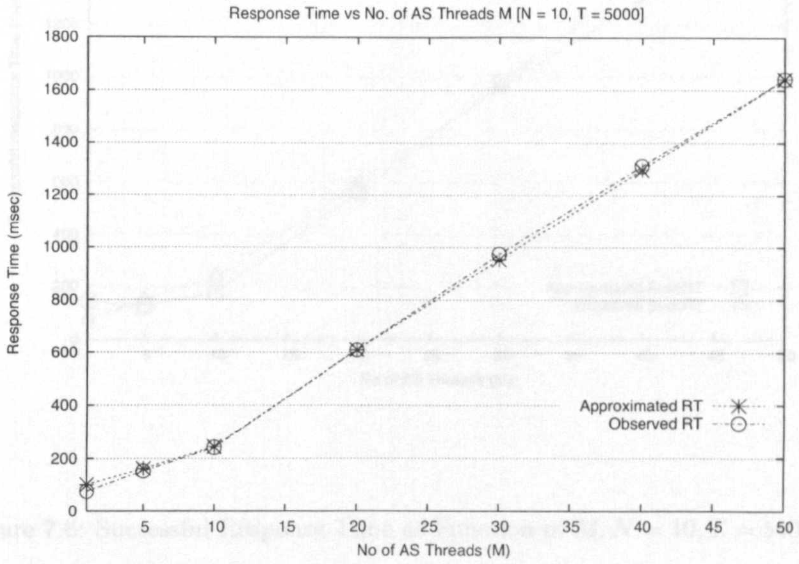


Figure 7.5: Response Time as Function of $M, N = 10, T = 5000$

In the first set of experiments, M is varied in the range $[1, \ldots, 50]$, while N is kept constant at $N = 10$. This configuration is the same as the one used in the first set of simulation experiments, except for the static parameters that are derived from the calibration process.

In Figure 7.5 the average response time is plotted against M. As we can see, the response time estimated by the approximation ($ApproximatedRT$) remains nearly identical to the response time observed on the real system ($ObservedRT$) and both increase with M. This is clearly due to increased concurrency as more threads, each with a job, compete for the same resources, slowing down the performance.

Figure 7.6 shows the average Successful Response Time that takes into consideration only those jobs which transactions end with a commit, no rejection at the application server queue or at the database queue is included. As we can see the Approximated and Observed values are very close, even if not uniformly, but with the plots criss-crossing each other.

The same match between Approximated and Observed values can be seen in Figure

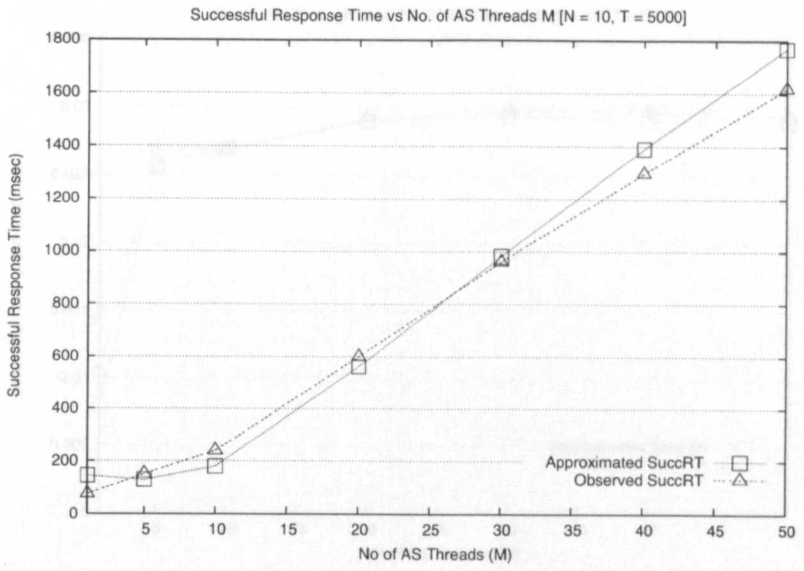


Figure 7.6: Successful Response Time as Function of $M, N = 10, T = 5000$

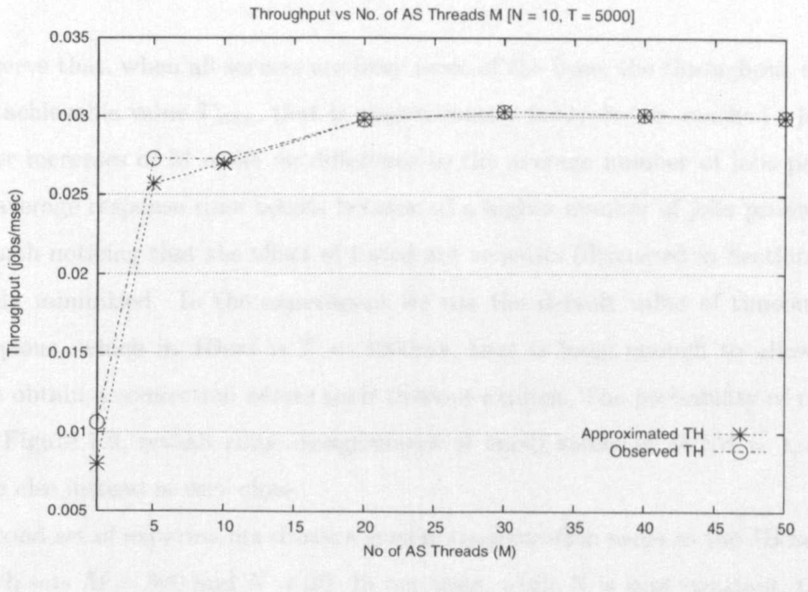


Figure 7.7: Throughput as Function of $M, N = 10, T = 5000$

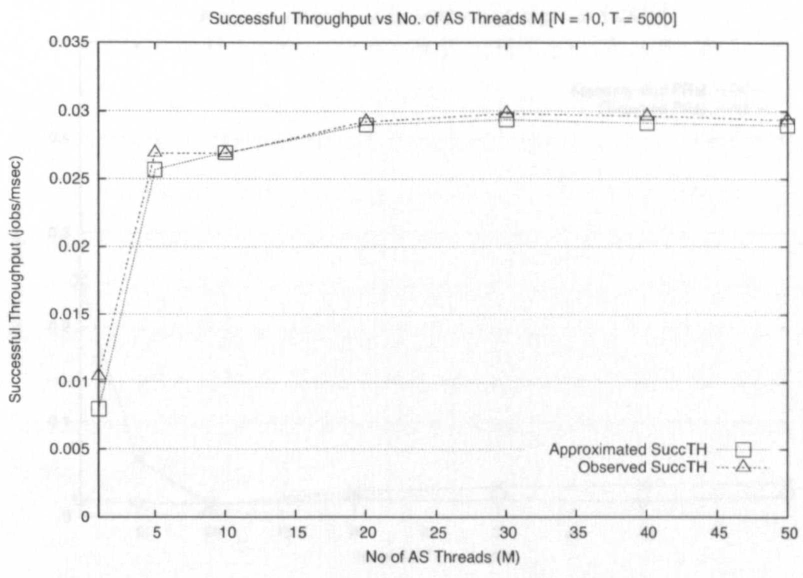


Figure 7.8: Successful Throughput as Function of $M, N = 10, T = 5000$

7.7 and Figure 7.8 representing the average throughput and successful throughput, respectively.

We observe that, when all servers are busy most of the time, the throughput reaches a maximum achievable value T_{max}, that is approximately $0.03 jobs/sec$ reached when $M = 30$. Further increases of M make no difference to the average number of jobs processed, while the average response time boosts because of a higher number of jobs present.

It is worth noticing that the effect of timed out requests (discussed in Section 7.3.1.4) is effectively minimized. In the experiment we use the default value of timeout at the database queue, which in JBoss is $T = 5000ms$, that is large enough to allow all the requests to obtain a connection before their timeout expires. The probability of rejection, plotted in Figure 7.9, reveals some disagreement at small values of M ($M = 1, M = 5$), everywhere else instead is very close.

The second set of experiments shows a system configuration same as the JBoss default value, which sets $M = 300$ and $N = 20$. In our tests, while N is kept constant, the value of M is varied in the range $[100, \ldots, 300]$.

The values of approximated and observed response time illustrated in Figure 7.10 are once more very close. Here we observe again that the response time progressively increases as the number of server threads increases, this trend is gradual up to $M = 200$, while it

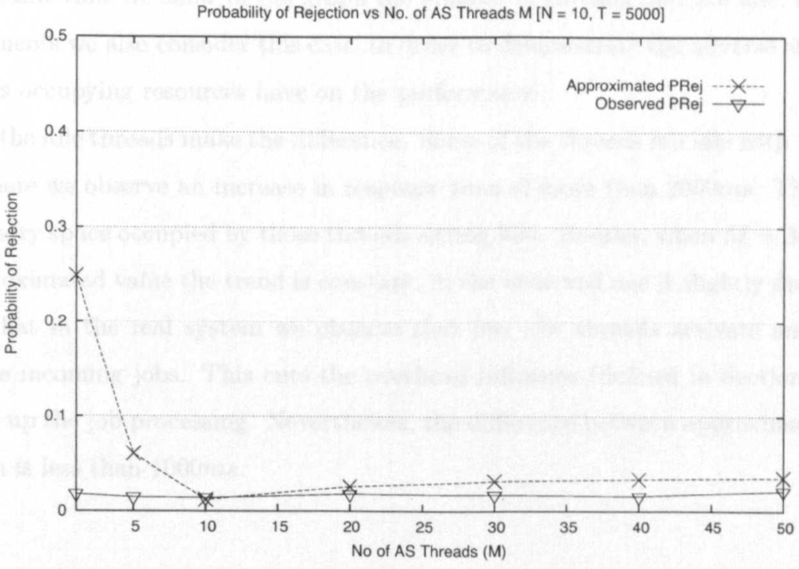


Figure 7.9: Probability of Rejection as Function of $M, N = 10, T = 5000$

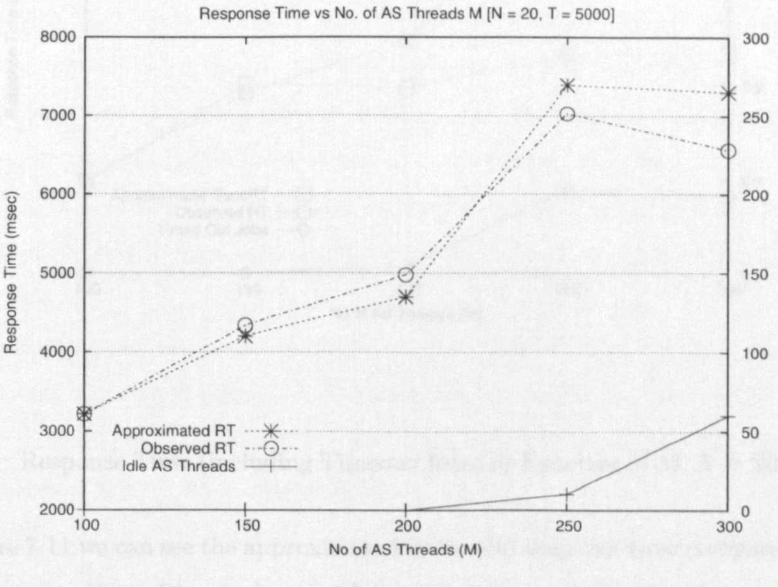


Figure 7.10: Response Time as Function of $M, N = 20, T = 5000$

boosts in the range $[200, \dots, 250]$ and then stops when $M = 250$.

At the same time we show in the graph the number of threads that are idle, if any. In our experiments we also consider this case, in order to demonstrate the adverse effect that idle threads occupying resources have on the performance.

Indeed the idle threads make the difference. Some of the threads are idle with $M = 250$ and just there we observe an increase in response time of more than $2000ms$. This is due to the memory space occupied by those threads sitting idle. Besides, when $M = 300$, while in the approximated value the trend is constant, in the observed one it slightly drops. The reason is that in the real system we observe that few idle threads activate and accept some of the incoming jobs. This cuts the overhead influence (defined in Section 7.3.1.2) and speeds up the job processing. Nevertheless, the difference between approximation and observation is less than $1000ms$.

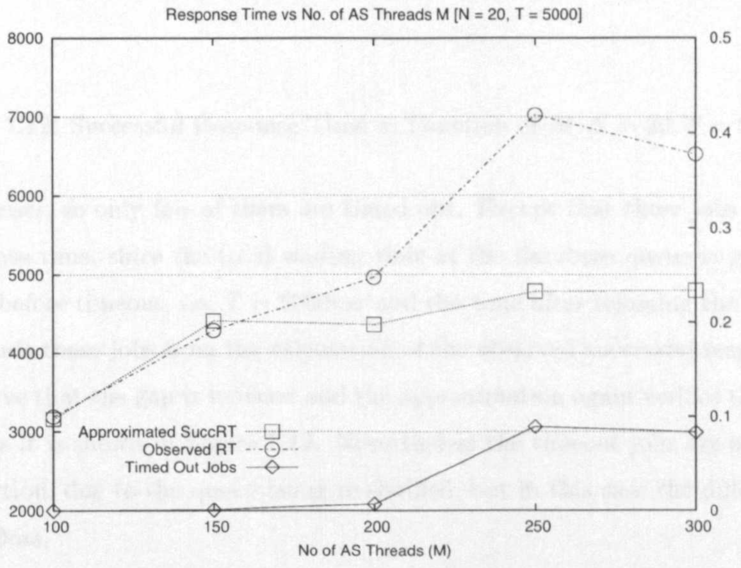


Figure 7.11: Response Time (including Timeout Jobs) as Function of $M, N = 20, T = 5000$

In Figure 7.11 we can see the approximated successful response time compared with the observed response time. Observed successful response time sets higher than approximated because timed out jobs are inserted back into the queue and the timeout delay adds up to the response time. While, as explained in the previous section, in our model those jobs are rejected straight away.

However, the ratio of timeout requests is approximately 0.08 over the total number of

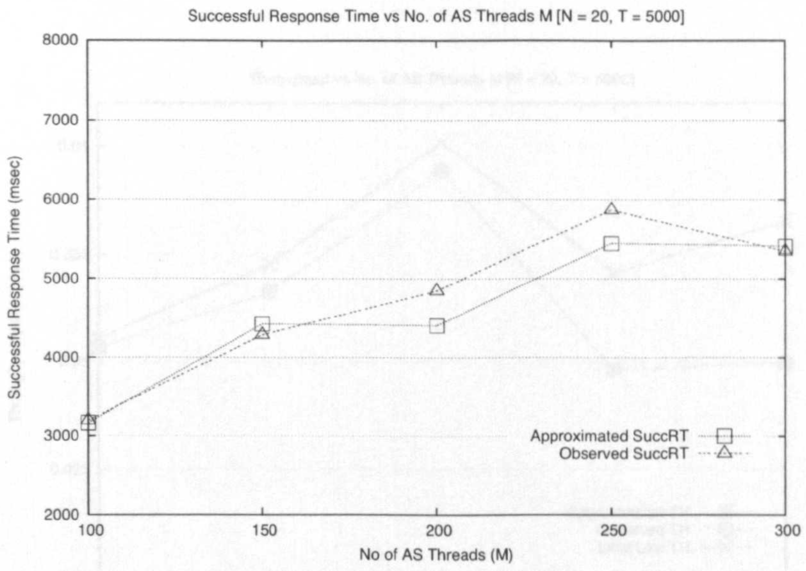


Figure 7.12: Successful Response Time as Function of $M, N = 20, T = 5000$

database accesses, so only few of them are timed out. Except that these jobs experience a large response time, since the total waiting time at the database queue is made of the waiting time before timeout, i.e. $T = 5000ms$ and the time after rejoining the queue.

If we exclude those jobs from the calculation of the observed successful response time, then we observe that the gap is reduced and the approximation again verifies the observation values, as it is shown in Figure 7.12. Nevertheless the timeout jobs are still causing a small distortion, due to the queue being re-shuffled, but in this case the difference is of less than $1000ms$.

The throughput and successful throughput plots, presented in Figure 7.13 and Figure 7.14 respectively, show how close the estimates of the approximation value are to the reality of the observation value. Here again the idle server threads are making the difference, as previously seen for the response time. In Figure 7.10 we observed a boost in response time when $M > 200$, at the same point here we observe a collapse in number of jobs processed per unit of time. The performance is degrading because of the memory space occupied by those threads sitting idle.

In Figure 7.13 we also notice the influence of the overhead, described in Section 7.3.1.2, with the discrepancy increasing as the number of server threads increases. If the overhead were null, then the resulting throughput, named here *Little's Law TH*, would be higher than the one actually achieved.

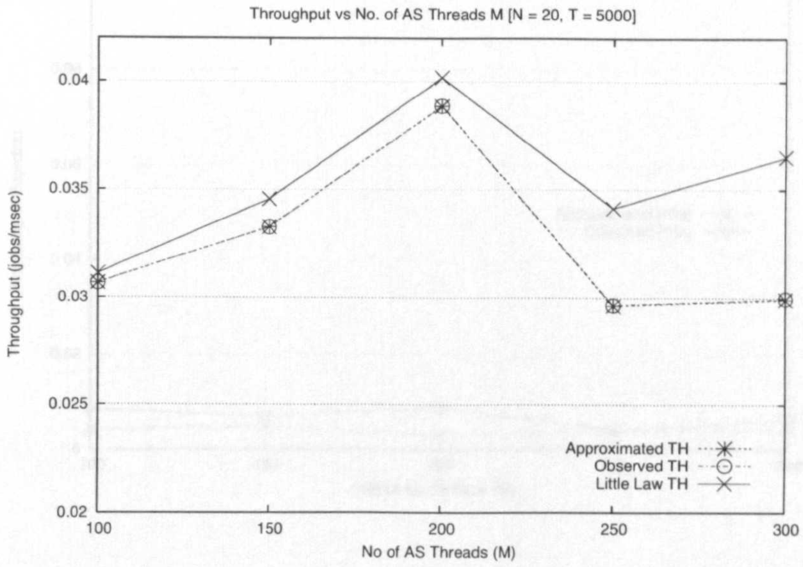


Figure 7.13: Throughput as Function of $M, N = 20, T = 5000$

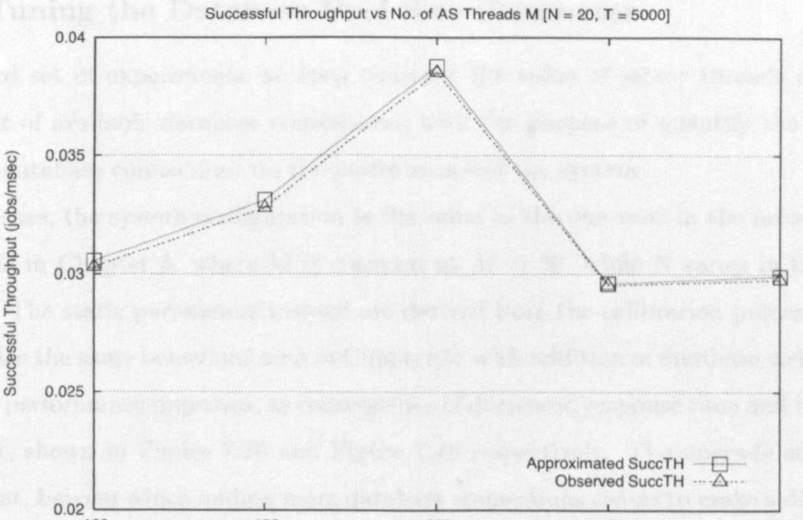


Figure 7.14: Successful Throughput as Function of $M, N = 20, T = 5000$

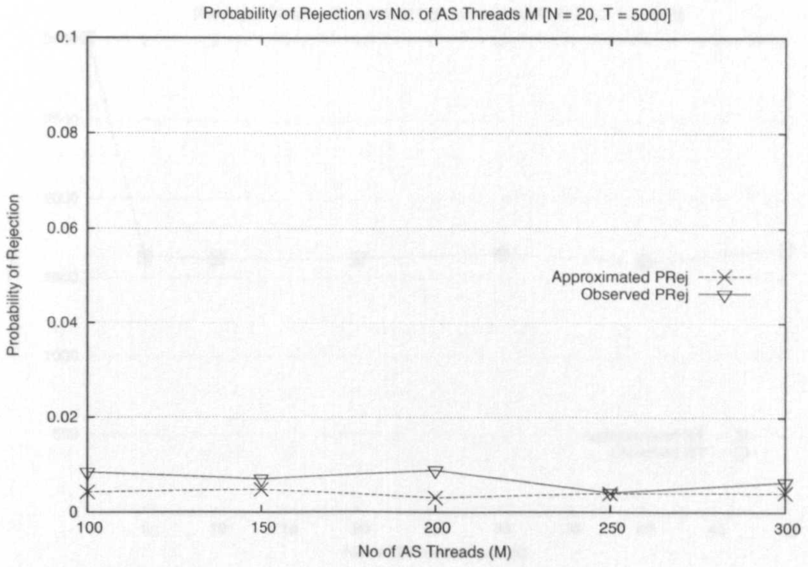


Figure 7.15: Probability of Rejection as Function of $M, N = 20, T = 5000$

The probability of rejection shown in Figure 7.15 confirms the validation of the approximation.

7.4.2 Tuning the Database Pool Size Parameter

In the third set of experiments we keep constant the value of server threads and vary the number of available database connections, with the purpose of quantify the effect of number of database connections on the performance of the system.

In this case, the system configuration is the same as the one used in the second set of experiments in Chapter 5, where M is constant at $M = 50$, while N varies in the range $[1, \ldots, 50]$. The static parameters instead are derived from the calibration process.

We notice the same behaviour seen in Chapter 5: with addition of database connections the system performance improves, as consequence of decreased response time and increased throughput, shown in Figure 7.16 and Figure 7.18 respectively. The upgrade stops at a certain point, beyond which adding more database connections ceases to make a difference. For the present configuration this point is $N = 5$, but increasing the number of server threads, as we observe in the final set of experiments, the optimum number of connections become larger. The successful throughput in Figure 7.19 shows again the accuracy of the approximate solution.

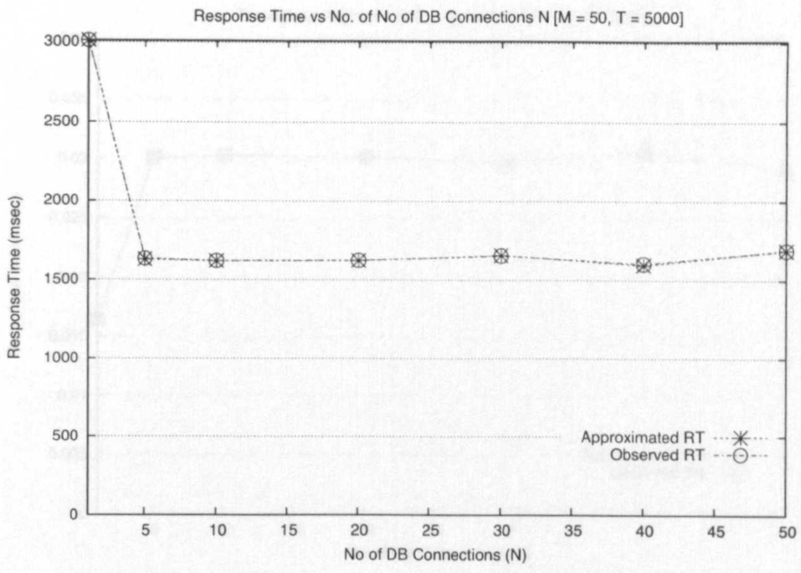


Figure 7.16: Response Time as Function of $N, M = 50, T = 5000$

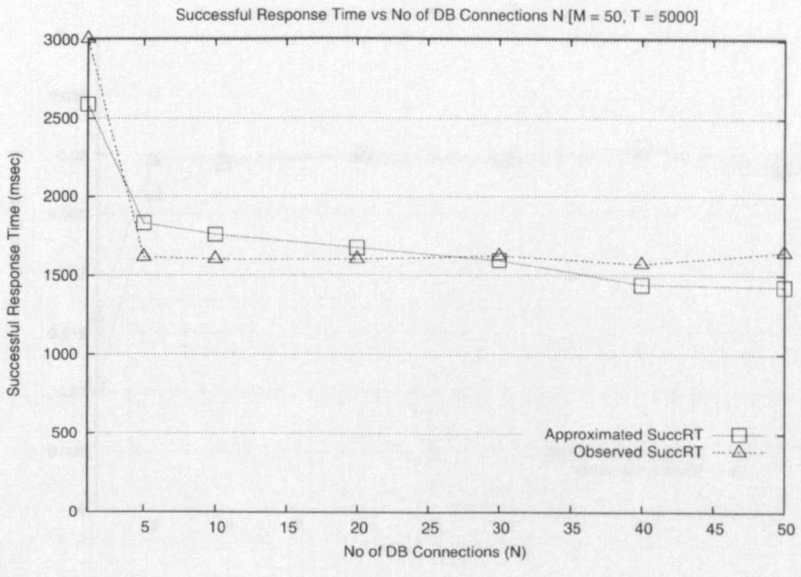


Figure 7.17: Successful Response Time as Function of $N, M = 50, T = 5000$

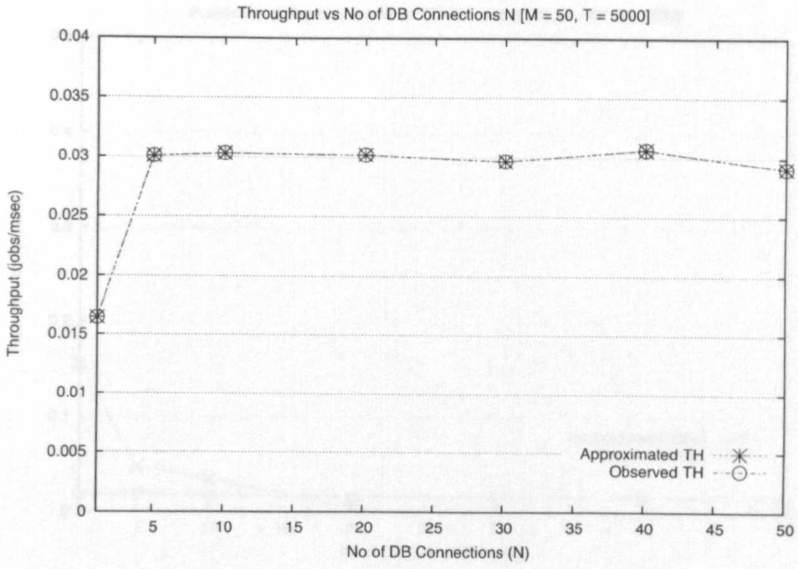


Figure 7.18: Throughput as Function of $N, M = 50, T = 5000$

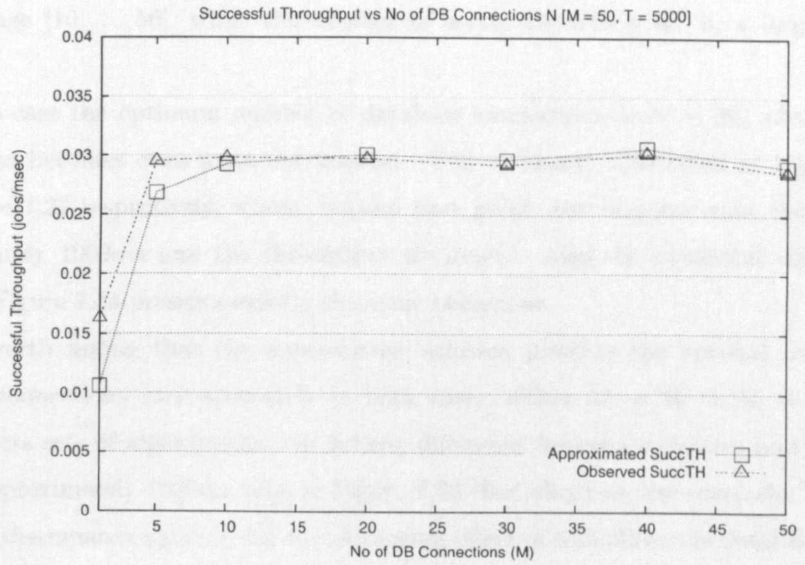


Figure 7.19: Successful Throughput as Function of $N, M = 50, T = 5000$

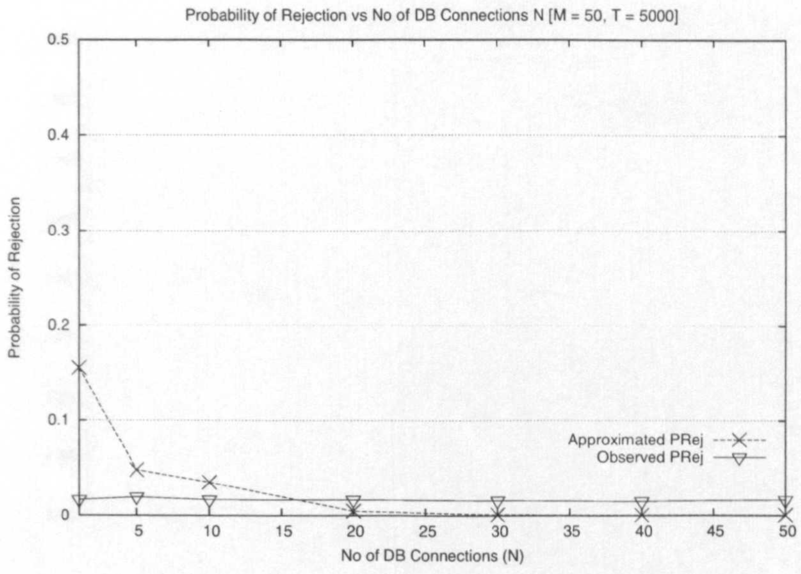


Figure 7.20: Probability of Rejection as Function of $N, M = 50, T = 5000$

The final set of experiments confirms the previous observations. Here again N is varied in the range $[10, \ldots, 50]$, while the number of server threads is set to a larger value, $M = 200$.

In this case the optimum number of database connections is $N = 30$, adding more connections becomes even more detrimental. This is clearly illustrated in Figure 7.21 and Figure 7.23 respectively, where, beyond that point, the response time increases of approximately $1000ms$ and the throughput decreases. Also the successful throughput shown in Figure 7.24 presents exactly the same behaviour.

It is worth noting that the approximate solution predicts the optimal number of database connections very accurately in both cases, either $M = 50$ or $M = 200$. In the two lasts sets of experiments, the largest difference between estimates and observations is approximately $1000ms$ seen in Figure 7.22 that illustrate the successful response time. The discrepancy again is due to the timeout effect of reshuffling the database queue, it is negligible instead when $M = 50$, as illustrated in 7.17. The same timeout effect disturbs the probability of rejection graphs, in Figure 7.20 and Figure 7.25.

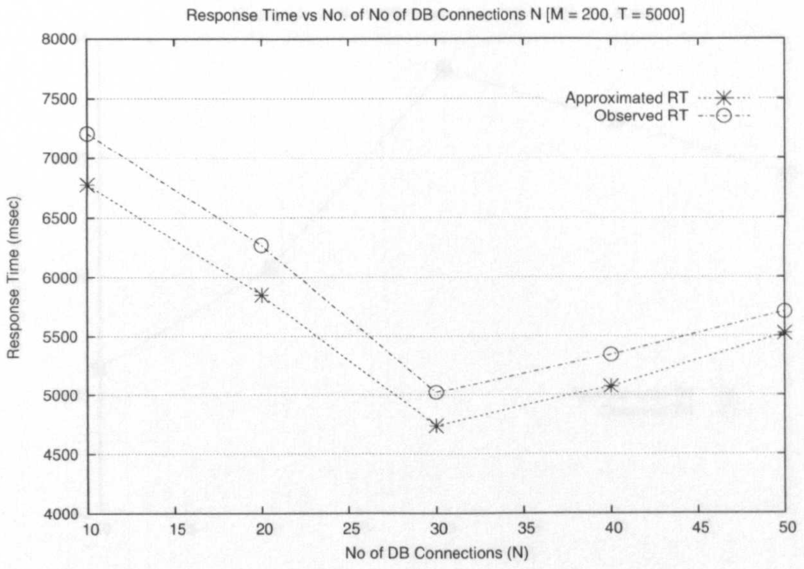


Figure 7.21: Response Time as Function of $N, M = 200, T = 5000$

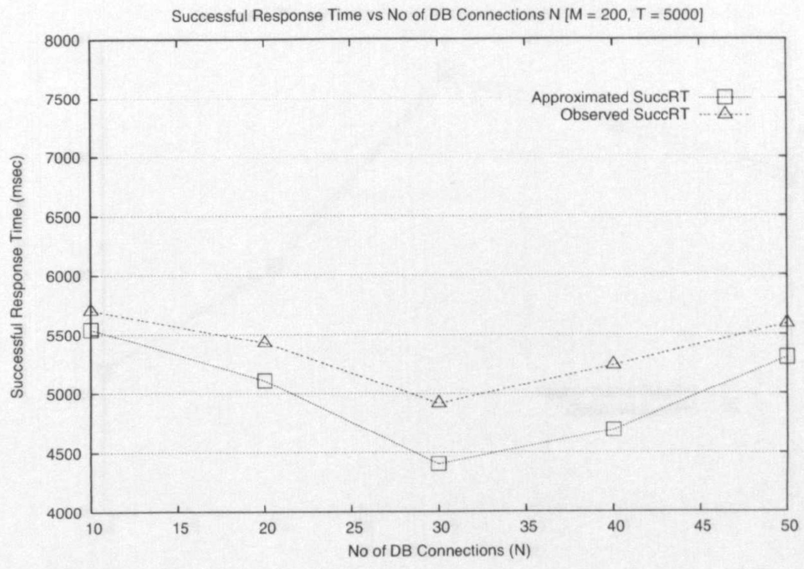


Figure 7.22: Successful Response Time as Function of $N, M = 200, T = 5000$

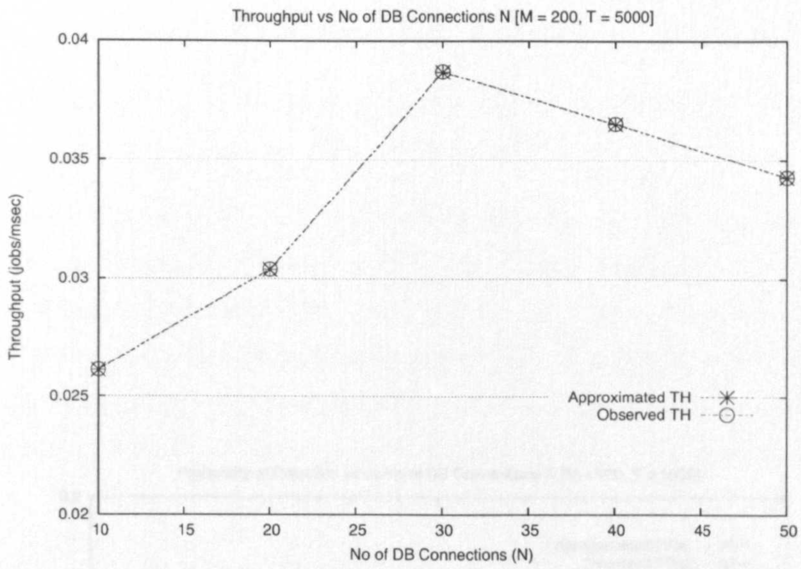


Figure 7.23: Throughput as Function of $N, M = 200, T = 5000$

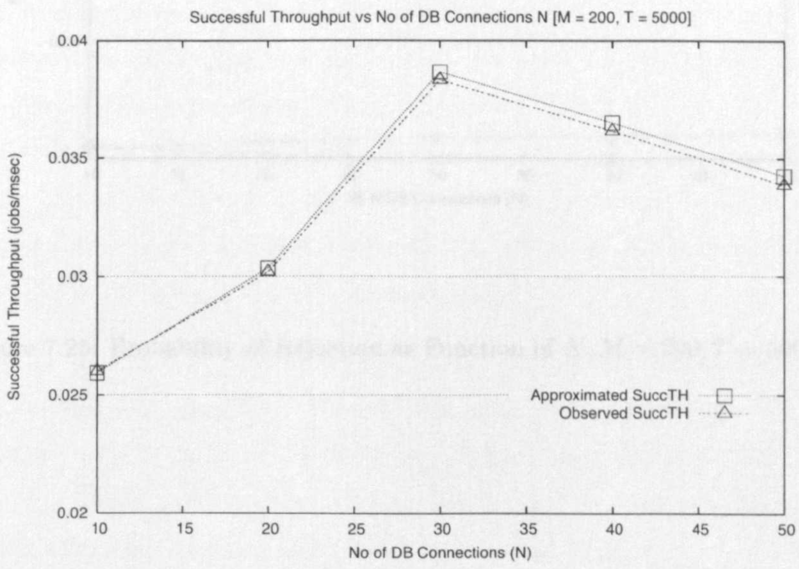


Figure 7.24: Successful Throughput as Function of $N, M = 200, T = 5000$

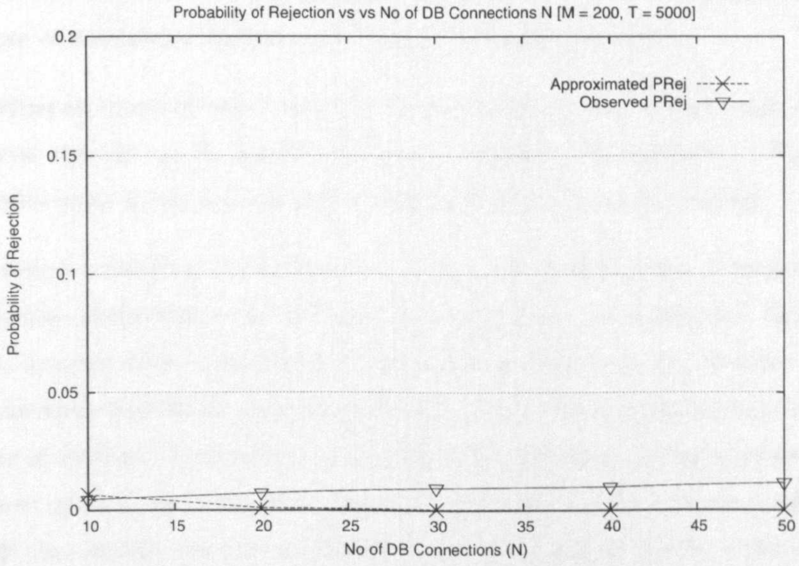


Figure 7.25: Probability of Rejection as Function of $N, M = 200, T = 5000$

7.5 Summary of Observations

In summary, the following significant observations can be made after the experiments.

- It is possible and also beneficial to optimally tune the target system. The optimal tuning is a balance among the different configuration parameters.

- There is an optimal number of database connections that can be used and this depends on the number of application server threads. For instance, in the configuration set-up of Figure 7.7 we show that, if the number of database connection is set low ($N = 10$) the number of application server threads is optimally set to $M = 30$. Above this value the throughput remains flat, since the average number of processed jobs is constant, while the average response time increases linearly because of a higher number of concurrent threads competing for the same resources.

- Providing an excess of resources is not always advantageous. In particular, a surplus of server threads can be detrimental for performance. For instance, in Figure 7.10 we notice some server threads sitting idle and slowing down the system.

- Also over-provisioning of database resources is not recommended. A bigger number of database connections can improve performance up to a certain point. Beyond this point, opening more connections becomes detrimental, since the average response time increases and the throughput decreases. This estimate is particularly important in case of database server shared among different application servers in a cluster, since it allows optimal allocation of database resources. For instance, in our configuration set-up the optimal number of database connections is set to $N = 30$. If more connections are provided response time and throughput start to deteriorate, as it is shown in Figure 7.21 and Figure 7.23 respectively.

- The Timeout Effect, illustrated in Section 7.3.1.4, influences our estimate since the database queue is re-shuffled. However the impact can be minimized by using a high setting for the timeout value. In our experiments we use the JBoss default value, $T = 5000ms$ and we observe that a small ratio of database requests timeout. Besides, JBoss best practice recommends even higher settings.

- The experiments enhance the simulated results in Chapter 5. There again we observed that a small number of threads can limit the performance of the application server by serializing much of the processing, whereas a big number can consume resources and increase contention, again reducing the server performance.

- The model estimates of average response time, throughput and successful throughput are almost exact; there are some discrepancies in estimating successful response time and probability of rejection, mainly due to the Timeout Effect.

- In the overall, the approximation solution predicts the optimal system configuration very accurately. Therefore, there is no need of time consuming experiments running workload tests on a real system, our analytical approximation can be used to optimize the system configuration without employing the experimental approach. Once the model is calibrated, it is possible to simply use our analytical approximation (a) to carry out the capacity planning process of a real system, (b) to estimate the performance trend and (c) the optimal configuration setup for a given load and a given set of resources.

Chapter 8

Conclusions and Future Work

This thesis has addressed the issue of estimating and controlling performance of E-business systems. An analytical model has been designed to evaluate performance in terms of measurable and controllable parameters. By means of simulation, the approximate solution derived from the model has been proven to be accurate. A working solution for a QoS Control system has been provided. This system monitors and collects the key parameters of the model and controls the configuration setup of a real world E-business system. Experimental results have demonstrated that the approximation solution is effective in estimating the performance of a real E-business system and it precisely predicts the optimal configuration set-up.

In this chapter we summarise the achievements and we also propose some potential developments for future research.

8.1 Summary of Achievements

The main objective of this thesis is to design a performance model for estimating and controlling QoS of E-business systems.

To approach this, we have analyzed the activities within an E-business system by representing these in an abstract yet realistic manner. We have identified the various aspects that influence the system performance and we have postulated a set of rules which govern the interaction between the different components.

The outcome has been a model that allows evaluation of performance metrics in terms of measurable or controllable parameters. The model captures the behaviour of a large class of E-business systems, since it is not tied to a specific technology. Besides, it is sufficiently detailed to take into account the essential system features and simple enough to be analytically and numerically tractable.

Through analysis of the model, we have obtained an usable and efficient approximate solution that returns the QoS metrics of interest. The accuracy of this solution has been evaluated by comparing the model estimates to those obtained from simulations. Furthermore, we have examined the effect of several controllable parameters on the performance of the system with a series of numerical and simulation experiments. We have observed that:

- The approximate solution is almost exact in predicting the performance of the system, with some discrepancies shown only close to the saturation point of the system.
- The approximate solution tends to be pessimistic, overestimating the response time and underestimates the throughput.
- The model is more accurate in predicting the successful throughput than the total one.
- From the performance point of view, an optimal number of thread exists and it depends on the average service times at the CPU, disk and database servers.
- The system performance improves with the addition of database servers, but only up to a point. Beyond that, adding more database service capacity ceases to make a difference and the extra servers are not utilized.

The subsequent step has been to investigate to what extent the model is matching the reality, comparing the estimated performance to those observed on an E-business system of realistic complexity.

The E-Business system that we have considered for the observations is constituted by a J2EE application (the *ECperf* benchmarking application), the JBoss application server and a database server. On the purpose of quantifying both the static parameters of the model and the observed performance, we have implemented a QoS Control system which monitors the JBoss application server and controls its configuration parameters.

Then we have calibrated the model to assess the correspondence of the model with a real system. We have run several sets of experiments in order to observe the behaviour of a real E-business system and to evaluate the static parameters. These parameters have been substituted in the approximate solution and the estimated performance has been compared to the observed ones. The comparison shows that the approximate solution is almost exact in predicting the performance and it assesses the optimal system configuration very accurately.

The solution we have found can be used in order:

- to predict the performance of a system under given or assumed loading conditions, establishing operating baselines and trends;

- to determine the capacity to allocate so that peak workloads can be serviced;
- to identify system bottlenecks;
- to choose optimal settings for certain controllable parameters with respect to specified performance measures.

8.2 Future Work

In this section we propose some potential development that could stem out from our work.

8.2.1 Self-Adaptive Single Machine Controller Unit

The QoSCU, described in Chapter 6, can be extended to become a Self-Adaptive Control Unit.

The model will be used as analytical foundation for the QoSCU running on the Target System described in Chapter 7. The QoSCU will be extended to include the approximate solution and compute the optimum configuration set-up at run time, while data from the monitor subsystem are collected. The Control Subsystem will periodically compare the actual performance to the expected ones. If any violation is detected, the system will improve its performance with an automatic fine tuning of the application server, using the approximate solution to determine the best configuration with respect to specified performance measures.

A set of optimal system configuration could be provided, or it could be gradually discovered by means of experiments, for instance extending the Testing Phase to include more configuration setup. Also the set of optimal configurations could be provided by an algorithm, such as the *Hill Climbing Algorithm* adopted by [38]. With this algorithm, from the default configuration, all neighbour configurations are examined (a neighbour configuration is dened as the configuration in which the value of one of the parameters is incremented by moving one step). For every possible neighbour configuration, a local function is estimated to compute the achieved QoS. The configuration with the largest QoS is selected as the next one to examine and the process repeats itself from that configuration. The search continues until either no improvement can be made or a limit on the maximum number of hops is reached.

8.2.2 Self-Learning (Autonomic) Controller Unit

It is possible to enhance the Controller Unit from a computational controller, which monitors the performance and computes the best parameter configuration, to a self-learning

Controller Unit, which will memorize the different conditions it experiences and the strategies it applies. In future, if some conditions will repeat, the same strategy will be applied again without any further computation. With this approach the system will become able to self learn the strategies used time by time, recording in a database table all the different cases.

This type of system can be classified as *Autonomic System*, which monitors, experiments with, tunes its own parameters and learns how to make appropriate choices about keeping functions, just *"as muscles become stronger through exercise and the brain modifies its circuity during learning"*[27].

8.2.3 Single Machine Controller Unit in a Cluster

The model could also be extended to describe a cluster of nodes. With regard to the Data Tier, traditionally it employs a *shared-nothing* architecture which does not support replication. Whereas at the Business Tier, the cluster can be made up of a set of nodes, each of them representing an instance of the application server engine,

On the basis of the logical tiers mentioned in Chapter 3, we have three main architectures for a cluster [60].

Figure 8.1 illustrates the first architecture, *Nodes Hosting Both WS and AS*, where a specific load balancer is used to distribute the load between the cluster nodes, which host both the web server (WS) and the application server (AS) tiers. Load balancers come in different flavours, hardware based, software based, or a combination of the two. Load balancing policies, such as *Round Robin*, are the algorithms used to implement the distribution of the requests among the nodes.

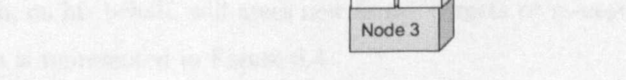


Figure 8.1: Cluster of Nodes Hosting Both WS and AS

Figure 8.2 illustrates the second architecture, *Cluster of AS*, where a single web server is located in a different node in respect of the cluster of application servers. The web server here acts as single points of entry into the cluster and as traffic directors to individual application servers for the business logic computation.

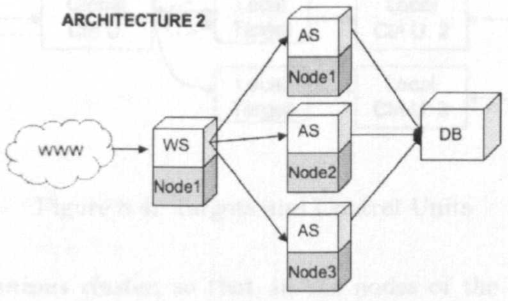


Figure 8.2: Cluster of AS

Figure 8.3 illustrates the third architecture, *Cluster of WS and cluster of AS*, where the load balancer distributes the load between a cluster of web servers, which in turn redirect the traffic to a cluster of application servers. This case is the combination of the previous two architectures, therefore the same considerations valid there, will hold true.

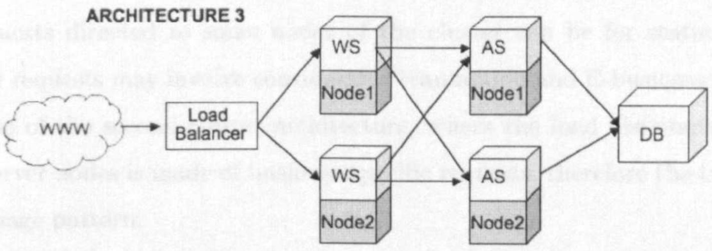


Figure 8.3: Cluster of WS and Cluster of AS

The E-business system has to grant that a number of QoS guarantees are met in the provisioning of the services. Let's consider the metrics to meet as the Total Target of the system. The Global Controller Unit will work out the control actions that are carried out in order to maintain the target. In the case of a cluster, the Global CU will divide the target in Local Targets that must be maintained by the Local CU of the cluster. It may happen that a Local CU will not bear the load, therefore it will solicit a lower target to the Global CU, which, on his behalf, will asses new Local Targets or re-negotiates the Global Target. This idea is represented in Figure 8.4

The analytical foundation of the Controller Unit has been designed for a single instance of an application server node, connected to a single instance of database. This represents a special case, i.e. when the Total Target matches the Local Target and the Global CU coincides with the Local CU. The single instance model can be easily extended to the cluster under the following conditions:

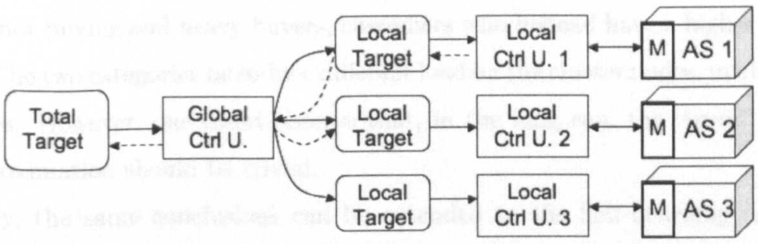


Figure 8.4: Targets and Control Units

- If it is a *homogeneous cluster*, so that all the nodes of the cluster have the same hardware characteristics;
- if the cluster is *uniformly managed* (or has a "homogeneous deployment"), so that all the nodes provide the same services.

Under these conditions, the approximations to consider are concerned with *load* and *session*.

The approximation concerning load should be applied to the first cluster architecture, since the requests directed to some nodes of the cluster can be for static content only whereas other requests may involve considerable transaction and E-business activity. This is not the case of the second cluster architecture, where the load distributed among the application server nodes is made of business-specific requests, therefore the traffic presents an uniform usage pattern.

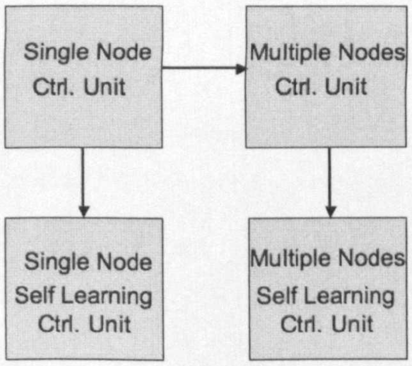

Single Node Ctrl. Unit
Multiple Nodes Ctrl. Unit
Single Node Self Learning Ctrl. Unit
Multiple Nodes Self Learning Ctrl. Unit

Figure 8.5: Potential Evolution Process

The approximation concerning session occurs because the access to a web service is commonly organized in form of *client session*, consisting of many individual requests, such as browse, search, select and pay. An analysis of the customer behaviour [39] shows that different categories of clients exists. For instance, considering the case of a web store, there are occasional customers, clients who simply search for information on existing products but end up not buying and heavy buyers, customers who instead have a higher probability of buying. The two categories introduce different load on the cluster nodes, in all the cluster architectures. However, one could observe that, in the long run, the "error" introduced by the approximation should be trivial.

Naturally, the same conclusions can be extended to the Self-Learning (Autonomic) Control Unit, which self-learning functions can be enhanced to include the cluster nodes.

The potential development that we suggest are summarized in Figure 8.5, which illustrates the overall evolution process.

Bibliography

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[2] D. Bacigalupo, S. Jarvis, L. He, and G. Nudd. An investigation into the application of different performance prediction techniques to e-commerce applications. In *Workshop on Performance Modelling, Evaluation and Optimization of Parallel Distributed Systems, in conjuction with IPDPS04*, New Mexico, USA, April 2004.

[3] D. Bacigalupo, S. Jarvis, L. He, D. Spooner, and G. Nudd. Comparing layered queuing and historical performance models of a distributed enterprise application. In *nternational Conference on Parallel and Distributed Computing and Networks (PDCN'05)*, Innsbruck, Austria,, February 2005.

[4] BEA Weblogic. http://www.bea.com.

[5] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, 1999.

[6] P. Brebner, S. Chen, I. Gorton, J. Gosper, L. Hu, A. Liu, D. Palmer, and S. Ran. Report: Evaluating j2ee application servers. Technical report, CSIRO Middleware Technology Evaluation Series, 2002.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 246–261. ACM Press, 2002.

[8] L. Cherkasova and P. Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Trans. Comput.*, 51(6):669–685, 2002.

[9] S. Deshpande and B. Martin. Eight reasons ecperf is the right way to evaluate j2ee performance.

[10] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Network Operations and Management*. IEEE, 2002.

[11] J. C. Doyle, B. A. Francis, and A. R. Tannenbaum. *Feedback control theory*. MacMillan, New York, 1992.

[12] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in dynamic e-commerce web sites. In *International World Wide Web Conference*, pages 276–286, May 2004.

[13] G. Ferrari, S. Shrivastava, and P. Ezhilchelvan. An approach to adaptive performance tuning of application servers. In *1st IEEE International Workshop on QoS in Application Servers, in conjunction with SRDS04*, Brasil, October 2004.

[14] M. Fleury and F. Reverbel. The jboss extensible server. In M. Endler and D. Schmidt, editors, *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 344–373. Springer-Verlag, 2003.

[15] IBM. Trade Benchmark Application. http://www.ibm.com/software/websphere.

[16] IBM. Websphere. http://www.ibm.com/software/websphere.

[17] IBM. *Websphere Application Server, Advanced Edition Tuning Guide*, 2005.

[18] JBoss. http://jboss.com.

[19] JBoss. JBoss Managed Connections - Wiki pages. http://wiki.jboss.org.

[20] JBoss. JBoss Messaging. http://labs.jboss.com/portal/jbossmessaging.

[21] JBoss. JBossMQ. http://www.jboss.org/wiki/page=jbossmq.

[22] JBoss. Pooled Invoker - Wiki pages. http://wiki.jboss.org/wiki/page=pooledinvokerconfig.

[23] G. Jung, G. Swint, J. Parekh, C. Pu, and A. Sahai. Detecting bottleneck in n-Tier IT applications through analysis. In *IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006)*, Dublin, Ireland, October 23-25 2006.

[24] A. Kamra, V. Misra, and E. Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *IWQoS*, pages 47–56, Montreal, Canada, June 2004.

[25] K. Kang, S. Son, and J. Stankovic. Differentiated real-time data services for e-commerce applications. In *Electronic Commerce Research*, volume 3(1-2), pages 113–142, 2003.

[26] M. Karlsson, K. Moore, E. Hagersten, and D. Wood. Memory characterization of the ecperf benchmark. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI 2002), held in conjunction with the 29th International Symposium on Computer Architecture (ISCA29)*, Alaska, USA, May 2002.

[27] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[28] A. I. Kistijantoro, G. Morgan, S. K. Shrivastava, and M. C. Little. Component replication in distributed systems: a case study using enterprise java beans. In *22nd International Symposium On Reliable Distributed Systems (SRDS)*, Florence (Italy), 2003.

[29] S. Kounev and A. Buchmann. Performance modeling and evaluation of large-scale j2ee applications. In *Proc. of the 29th International Conference of the Computer Measurement Group on Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG2003)*, Dec. 2003.

[30] S. Kounev and A. Buchmann. Performance modelling of distributed e-business applications using queuing petri nets. In *Proc. of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03)*. IEEE, March 2003.

[31] S. Kounev, B. Weis, and A. Buchmann. Performance tuning and optimization of J2EE applications on the JBoss platform. In *In Journal of Computer Resource Management, Issue 113*, 2004.

[32] J. Lindfors and M. Fleury. *JMX - Managing J2EE with Java Management Extensions*. SAMS, 2002.

[33] M. Little, J. McGovern, R. Adatia, Y. Fain, J. Gordon, E. Henry, W. Hurst, A. Jain, V. Nagarajan, H. Oak, and L. Phillips. *Java 2 Enterprise Edition 1.4 (J2EE 1.4) - Bible*. Wiley, 2003.

[34] T. Liu, S. Kumaran, and Z. Luo. Layered Queueing Models for Enterprise JavaBean applications. In *Proceedings of the 5th IEEE international Conference on Enterprise Distributed Object Computing*, Washington, DC, September 2001.

[35] Y. Liu, S. Chen, I. Gorton, and A. Fekete. A methodology for predicting the performance of component based systems. Poster session of ACM ICS, May 2004.

[36] Y. Liu, A. Fekete, and I. Gorton. Design level performance modeling of component-based applications. Technical Report 543, School of Information Technologies, University of Sydney, 2003.

[37] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, June 2001.

[38] D. Menasce, D. Barbara, and R. Dodge. Preserving qos of e-commerce sites through self-tuning: a performance model approach. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, Florida, 2002. ACM.

[39] D. A. Menasce, V. Almeida, R. Fonseca, and M. Mendes. A methodology for workload characterization for e-commerce servers. In *ACM Conference in Electronic Commerce*, pages 119–128, Denver, CO, Nov. 1999.

[40] D. A. Menasce, V. Almeida, R. Riedi, F. Ribeiro, R. Fonseca, and W. Meria. In search of invariants for e-business workloads. In *Proceedings of the 2nd ACM conference on Electronic Commerce*, pages pp. 56–65, 2000.

[41] D. A. Menasce and V. A. Almeida. *Capacity planning for Web Services*. Prentice-Hall Inc., 2002.

[42] I. Mitrani. *Modelling of computers and communication systems*. Cambridge University Press, 1987.

[43] I. Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1998.

[44] C. Molina-Jimenez, S. Shrivastava, J. Crowcroft, and P. Gevros. On the monitoring of contractual service level agreements. In *IEEE International Workshop on Electronic Contracting (WEC)*, San Diego, July 2004.

[45] Newcastle University. TAPAS: Trusted and QoS-Aware Provision of Application Services. http://tapas.sourceforge.net/.

[46] R. Oliva. Model calibration as a testing strategy for system. *European Journal of Operational Research*, (151):552–568, 2003.

[47] Oracle. Oracle Application Server. http://www.oracle.com/appserver.

[48] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Syst.*, 23(1-2):127–141, 2002.

[49] M. Raghavachari, D. Reimer, and R. D. Johnson. The deployer's problem: Configuring application servers for performance and reliability. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 484–489, Washington, DC, USA, 2003. IEEE Computer Society.

[50] Rice University. RUBiS, http://rubis.objectweb.org.

[51] J. Spacco and W. Pugh. RUBiS Revisited: Why J2EE benchmarking is hard. In *In Companion To the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 2004.

[52] Standard Performance Evaluation Corporation (SPEC). SPECjAppServer01. http://www.spec.org/jAppServer.

[53] A. Stylianou, G. Ferrari, and P. Ezhilchelvan. A comparative evaluation of EJB implementation methods. In *10th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC07)*, Greece, May 2007.

[54] Sun Microsystems. EJB Specification http://java.sun.com/products/ejb.

[55] Sun Microsystems. J2EE - Java 2 Platform Enterprise Edition v 1.4 http://java.sun.com/j2ee.

[56] Sun Microsystems. JDBC. http://java.sun.com/products/jdbc.

[57] Sun Microsystems. JMS Java Message Service. http://java.sun.com/products/jms.

[58] Sun Microsystems. The Java Petstore 2.0. http://java.sun.com/developer/petstore.

[59] Sun Microsystems. *The ECperf 1.1 Benchmark Specification*, June 2001.

[60] R. Tyagi. Enterprise application clustering architecture, http://builder.com.com.

[61] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages pp. 291-302, 2005.

[62] U. Vallamsetty, K. Kant, and P. Mohapatra. Characterization of e-commerce traffic. In *the 4th IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS'02)*, June 2002.

[63] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. In *Proc. of IWQoS'04*, Montreal, Canada, 2004.

[64] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *the USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2003.

[65] B. Xi, Z. Liu, M. Raghavachari, C. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *International WWW Conference*, New York, 2004.

[66] B. Zeller and A. Kemper. Experience report: Exploiting advanced database optimization features for large-scale sap r/3 installations. In *the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, China, August 2002.

Appendix A

Octave Code

This appendix describes the code that implements the approximation solution. For this implementation we have used the GNU Octave, a high-level language for solving linear and nonlinear problems numerically. Three Octave functions constitute the implementation: *applic()*, *appDB()* and *appT()*.

```
function [ts,tu,w,sw,ps] = applic(k,q,lambda,xi,b,pr)
p = [zeros(1,k+q(1)),1];
t1 = zeros(1,k+q(1));
t0 = zeros(1,k+q(1));
sv = zeros(1,k+q(1));
j = min(q(1),[1:k+q(1)]);
for i=k+q(1):-1:1
  [t1(i),t0(i),sv(i)] = appDB([j(i),q(2)],xi,b,pr);
  p(i) = p(i+1)*(t1(i)+t0(i))/lambda;
end
p = p/sum(p);
L = p*[0:k+q(1)]';
t = lambda*(1-p(k+q(1)+1));
w = L/t;
ts = p([2:k+q(1)+1])*t1';
tu = p([2:k+q(1)+1])*t0';
ps = ts/lambda;
je = [1:k+q(1)] - j;
p = p/(1-p(k+q(1)+1));
p = p([1:k+q(1)]);
sw = (je/t + sv)*p';
```

Figure A.1: Octave Code for *applic*() Function

The Octave function *applic*() in Figure A.1 is the main function of the approximate solution. The input parameters are are:

- k, which is the external queue size;
- the vector $q[m, n]$, which provides m and n respectively the number of application server threads and database servers;

- the job arrival rate *lambda*;
- the timeout rate ξ at the database queue;
- the vector $b[b_0, b_1, b_{21}, b_{22}]$, where b_0 is the total CPU time consumed, b_1 is the disk time, b_{21} the database service time for non-transactional jobs and b_{22} service time for transactional jobs;
- the vector of probabilities $p[\alpha, \beta, \theta, \gamma]$, α is the probability for a job of leaving the CPU, β is the probability for a job of leaving the system, θ is the probability of being a transactional job, and γ is the probability of committing the transaction;

The output parameters are:

- the throughput of successful jobs, *ts*;
- the throughput of unsuccessful jobs, *tu*;
- the average response time, *w*;
- the average successful response time, *sw*;

Line 2-5: Initialize the vector of steady state probabilities, p, and the conditional successful, $t1$, the unsuccessful throughputs, $t0$, and rate of successful jobs.
Line 6-10: Being J the number of active jobs in the system, then the balance equations of Equation 4.8, are solved by calling the function *appDB*() shown in Figure A.2.
Line 11: The vector of probabilities is normalized, since they all add up to 1 ($p_0 + ... + p_J = 1$).
Line 12: Given the steady state probabilities by the vector p, it is possible to compute the average number of jobs in the system, as solved by the Equation 4.11.
Line 14: The average system throughput is equal to λ when the external queue is unbounded, otherwise, when the queue is bounded at level K, the throughput is computed by equation Equation 4.13.
Line 15: The average response time is obtained from Little's result, as it is denoted by Equation 4.15.
Line 16-17: The successful and unsuccessful throughputs are computed, considering that some of the database requests can timeout or some transactions can roll back, as it is defined by Equations 4.16 and 4.17
Line 18-19: When the external queue is bounded, some incoming jobs are rejected, in which case, the probability of rejection is defined by Equation 4.14
Line 19-21: The average successful response time is computed, considering the size of the external queue and the number of successful jobs.

```
function [t1,t0,sw] = appDB(q,xi,b,pr)
(2)  p = [zeros(1,q(1)),1];
(3)  if ((pr(2) == 1) | (pr(2) == 0))
(4)    tau = pr(1);
(5)  else
(6)    tau = pr(3)*pr(2)*log(1/pr(2))/(1-pr(2));
(7)  end
(8)  b2 = (1-tau)*b(3) + tau*b(4);
(9)  ser = min(q(1)-[0:q(1)],q(2));
(10) que = max(q(1)-q(2)-[0:q(1)],0);
(11) for i=q(1):-1:1
(12)   denom = ser(i)/b2 + que(i)*xi;
(13)   p(i) = p(i+1)*(1-pr(2))*appT(i,b(1),b(2),pr(2))/denom;
(14) end
(15) p = p/sum(p);
(16) t1 = 0;
(17) t0 = 0;
(18) tc = 0;
(19) for i=0:q(1)
(20)   thr1 = pr(2)*appT(i,b(1),b(2),pr(2)) + tau*pr(4)*ser(i+1)/b2;
(21)   thr0 = que(i+1)*xi + tau*(1-pr(4))*ser(i+1)/b2;
(22)   t1 = t1 + p(i+1)*thr1;
(23)   t0 = t0 + p(i+1)*thr0;
(24)   tc = tc + p(i+1)*appT(i,b(1),b(2),pr(2));
(25) end
(26) Lc = p*[0:q(1)]';
(27) wc = Lc/tc;
(28) qd = p*que';
(29) wd = qd/((1-pr(2))*tc);
(30) sw = (wc + (1-pr(2))*(wd/(1+xi*wd) + b2))/pr(2);
```

Figure A.2: Octave Code for *appDB*() Function

The Octave function *appDB*() in Figure A.2 is called by the main function *applic*() and is used to solve the balance equations in Equation 4.8. The input parameters are:

- the vector $q[m, n]$;
- the timeout rate ξ;
- the vector $b[b_0, b_1, b_{21}, b_{22}]$;
- the vector of probabilities $p[\alpha, \beta, \theta, \gamma]$, α;

The output parameters are:

- $t0$ throughputs of unsuccessful jobs;
- $t1$ throughputs of successful jobs;
- sw estimate of the average successful rsponse time;

Line 2: initialize vector of probabilities p_k

Lines 3 - 7: compute the fraction τ of database visits that are transactions. If the probability β of leaving the system is 0 or 1 then $\tau = \beta$, otherwise

$$\tau = \frac{\theta\beta}{1-\beta} \ln \frac{1}{\beta} .$$

as shown in Equation 4.5.

Line 8: The overall average database service time b_2, i.e. $1/\nu$, depends on the service time of database visits that are Transactions, b_{22}, and Non-Transactions, b_{21}, and it is given by Equation 4.6:

$$1/\nu = (1-\tau) * b_{22} + \tau * b_{21}$$

Line 9 - 14: At the steady state there are J active jobs in the system, the state of the network is described by k, which indicates the number of jobs at the computation subsystem, and $J - k$, that indicates the number of jobs at the database subsystem. The probability $p_k, (k = 0...J)$ that the network is in state k satisfies the balance equations in Equation 4.4, where:

- $min(J - k, n)$ is the number of jobs at the database, i.e. total number of busy database servers, where $J = m$ and $k = [0 : m]$ (for instance, if $m = 1$, $ser = [1, 0]$) (represented by vector at Line 9);
- $max(J - n - k, 0)$ is the number of jobs in the database queue (represented by vector at Line 10);
- t_k is the throughput of jobs leaving from the computation subsystem and it is returned by the function *appT*(), which solves the Equation 4.3, the related Octave function is shown in Figure A.3.

Line 15: The vector of probabilities is normalized.

Lines 16 - 25: The state-dependent throughput of the application server when it has J active jobs is obtained from Equation 4.7. This is composed by successful and unsuccessful throughput. The former is the average number of jobs that complete successfully, t_1, as it is computed in Equation 4.16. The successful throughput includes jobs leaving from the computation subsystem and transactions committed at the database. This is given by:

$$thr1 = \beta * t_k + \tau * \gamma * min(J - k, n)/b_2;$$

$$t_1 = thl + p_i * thr1;$$

The unsuccessful throughput, t_0, includes database requests that have timed out and database transactions that have rolled back. This is given by:

$$thr0 = max(J - n - k, 0) * \xi + \tau * (1 - \gamma) * min(J - k, n)/b_2;$$

$$th0 = th0 + p_i * thr0;$$

Line 26 - 30: The successful response time sw is given by:

$$sw = (wc + (1 - \beta) * (wd/(1 + \xi * wd) + b_2))/\beta$$

in which wc is the average time at the computation subsystem per visit, computed using Little's Law $wc = Lc/tc$, where Lc is the average number of jobs at the subsystem; and wd is the average waiting time at the database $wd = qd/((1 - \beta) * tc)$, where qd is the queue length at the database.

```
function [t] = appt(i,b0,b1,alpha)
(2)  r = (1-alpha)* b1/b0;
(3)  if (r == 1)
(4)    t = (alpha/b0)*i/(i+1);
(5)  else
(6)    t = (alpha/b0)*(1-r∧i)/(1-r∧(i+1));
(7)  end
```

Figure A.3: Octave Code for *appT*() Function

The input parameters are are:

- i, number of jobs present in the computation subsystem;
- $b0$, average CPU time per visit per job;
- $b1$, average disk time per visit per job ;
- $alpha$, probability for a job for having a database access;

The output parameter is the throughput of the computation subsystem

Line 1: Computes r, which is the ratio between the rate of departures and rate of arrival at the processor queue, as denoted by Equation 4.28.

Line 2-7: Returns the state-dependent throughput of the computation subsystem when there are k jobs in it, given by Equation 4.3.