Using Stochastic Petri Nets for Performance Modelling of Application Servers

Fábio N. Souza[1], Roberto D. Arteiro, Nelson S. Rosa, Paulo R. M. Maciel

Centro de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851 – 50740-540 – PE – Brasil
{fns,rda,nsr,prmm}@cin.ufpe.br

Abstract

Application servers have been widely adopted as distributed infrastructure (or middleware) for developing distributed systems. Current approaches for performance evaluation of application servers have mainly concentrated on the adoption of measurement techniques. This paper, however, focuses on the use of simulation techniques and presents an approach for performance modelling and evaluation of application servers using Petri nets. In order to illustrate how the proposed approach may be applied, Petri net models of JBoss application server are presented and their performance results are compared with ones that have been measured.

1. Introduction

Java 2 Platform, Enterprise Edition (J2EE) Specification defines a standard platform for supporting the development of distributed enterprise applications. Different implementations of J2EE specification (referred to as J2EE application servers) have been developed and are widely used as middleware platform in the corporative marketplace. Choosing the right implementation is a hard task as it involves evaluating many aspects such as cost, performance, scalability, flexibility and adaptability, ease of use, and standards conformance. However, that choice is usually focused on performance, since evaluating middleware based on other qualities is too hard [14].

Three different techniques can be used in performance evaluations: measurement, analytical modelling and simulation. The measurement approach has been widely used in the evaluation of performance of application servers. Cecchet [1] presents an analysis of the impact of application and container architectures in the overall performance of an Enterprise JavaBeans (EJB) system. Commonwealth Scientific and Industrial Research Organisation (CSIRO) [2] provides a detailed evaluation and comparison of different application servers presenting a qualitative analysis of its features and a quantitative analysis based on performance and stress tests performed using a stockbroking benchmark application. Koenev [6] uses SPECjAppServer2004 benchmark to explore the effect of different configuration parameters in performance of the JBoss Application Server (JBoss AS). Transaction Processing Performance Council (TPC) [13] specifies TPC Benchmark™ App (TPC-App), which is an application server and web services benchmark. Nevertheless, measurement techniques are very sensitive to variations in environment parameters. That limitation can compromise the accuracy of the results. Additionally, measurement experiments require building and configuring a separate environment. Costs associated with the necessary equipments, tools and time can be very high.

Analytic models can derive performance results quickly providing valuable performance information without having extra costs associated with the replication of the real environment. Lladó [8] presents an analytical model, based on queuing theory, which is suitable for performance analysis of EJB systems. Koenev [5] uses Non-Product-Form Queuing Networks to propose an analytical model for an application server running the SPECjAppServer2002 J2EE benchmark application. Liu [7] proposes an approach to predict the performance of applications running inside application servers at the design level. This prediction is based on a Queuing Network Model that is configured with parameters related to the workload as well as to the application server itself. However, analytical modelling requires so many simplifications and assumptions that is difficult to obtain accurate results [4]. Another problem concerning some analytical modelling tools is known as *the state-space*

[1] Supported by *Banco Central do Brasil*. The author hereby certifies that the content presented is of his own responsibility.

explosion, where the size of the model grows exponentially and memory resources quickly run out.

Once simulation models are built to run and not to be solved, they can incorporate more details (less assumptions) than analytical ones. So, those models can, theoretically, be more flexible and come closer to reality. Mc Guinness [11] presents a scalable simulation model of a multi-server EJB system that can be calibrated with parameters describing user interactions and server information. The outcome includes average execution time, throughput, and average CPU and I/O utilization. However, complex simulation models usually require a lot of time to execute. As each technique has its strengths and weaknesses, it is usually recommended using more than one of them to validate the performance results.

The main objective of this paper is to propose an approach to predict information about the performance of application servers through simulation models developed using Stochastic Petri Nets (SPNs). This formalism was chosen because it supports system performance evaluation, whilst it allows a natural modelling of various services offered by application servers (e.g. instance pool).

The remainder of this paper is organized as follows. Theory of Stochastic Petri Nets and basic concepts of application servers are briefly presented in Section 2. Section 3 presents the Petri net modelling approach used. Next, Section 4 presents the validation of the developed models using simulation and measurement techniques. Finally, concluding remarks and future work are presented in Section 5.

2. Background

2.1. Stochastic Petri Nets

Petri nets are a family of formal specification techniques that allows a graphical, mathematical representation and has powerful methods, which enable designers to perform qualitative and quantitative analysis. Place/transition Petri nets are used to model systems from a logical point of view, giving no formal attention to temporal relations and constraints. Generalised and Stochastic Petri Net (GSPN) [9] is one of the most extensively adopted classes of stochastic Petri nets. A GSPN is defined as a tuple $(P, T, I, O, H, G, M_0, W, \Pi)$ where: P is a set of places; T is a set of transitions; I, O and H are relations describing pre-conditions, post-conditions, and inhibition conditions, respectively; G is an enabling function that, given an immediate transition and a model state, determines if the transition is enabled or not; M_0 is a mapping from the set of places to the natural numbers describing the model initial state; W associates to each timed transition a non-negative real number, depicting the respective exponential transition delay (or rate), and to each immediate transition a non-negative real number representing its weight; and Π associates to each immediate transition a natural number that represents its priority level.

The set of places represents resources, local states and system variables. The set of transitions represents actions. This set is divided into two subsets: the set of immediate transitions that depicts actions that are irrelevant under the performance point of view; and the set of timed transitions, which have priority lower than immediate ones.

Deterministic and Stochastic Petri Nets (DSPNs) [10] are an extension to GSPN that includes the possibility of modelling transitions associated with a constant delay (deterministic transitions). A DSPN is a tuple $(P, T, I, O, H, G, M_0, \tau, W, \Pi)$, where P, T, I, O, H, G, M_o, and Π are defined as in GSPN. The function τ associates a non-negative real number to each timed transition, depicting the respective mean firing time (delay), while the function W associates a non-negative real number representing its weight to each immediate transition.

2.2. J2EE and Enterprise Java Beans

The core of J2EE specification is the definition of a framework for the development of server-side components known as Enterprise JavaBeans. EJB components (beans) are hosted in a runtime environment, named EJB container, which takes responsibility for managing their life cycle, managing resources in their behalf and providing them with predefined system-level services such as transaction management and security. To provide these services in a transparent way, an EJB container acts as a proxy between clients and beans.

One of the services provided by EJB containers is the *instance pooling*. When an EJB container intercepts a request, it may create a new bean instance to process that request. Otherwise, if there is an already created instance in memory, it may be better to reuse it, reducing the memory and time necessary to process the request. Sometimes, an EJB container may also reduce allocated resources by destroying bean instances that have not been used anymore. The actual mechanism used to instance pooling is not part of EJB specification, but depends on the EJB container implementation.

2.3. EJB in JBOSS

In JBoss architecture [3], a client application has access to an EJB component through a client-side proxy, which exposes the same methods that are implemented by the component itself. Each client-side proxy has a chain of *interceptors*. When a client application invokes one of the methods exposed by the proxy, it collects information about the invoked method in an object named *invocation* and delivers that object to the first interceptor in its chain.

This interceptor gathers some information about the context in which this invocation occurred (e.g. information about the user doing the request), adds that information to the invocation object and forwards it to the next interceptor in the chain. The last interceptor is in charge of dispatching the invocation object to the *invoker proxy,* which marshalls and forwards it through the network.

At the server side, an EJB container is created when an EJB component is deployed in JBoss application server. This container is associated with a server-side component named *invoker* whose role is to receive marshalled invocations, unmarshall and forward them to corresponding containers.

As any other EJB container, that one offered by JBoss is responsible for managing bean instances and providing them with predefined services. However, instead of implementing services itself, the container has a chain of pluggable server-side interceptors, each responsible for implementing a specific service (see Figure 1). When an EJB container receives an invocation object, it forwards that invocation to the first interceptor in its chain. This interceptor uses the information contained in the invocation object to perform the corresponding service and forwards the invocation to the next interceptor. Eventually, the instance interceptor (see Section 2.4) is invoked and obtains a bean instance to be used in the request processing. The last interceptor in this chain is responsible for identifying the method required, and invoking it using the obtained instance. The return of the method transverses the chain in the reverse way until been received by the container, which deliveries it back to the invoker. The invoker marshalls the returned value and sends it back to the invoker proxy, which unmarshalls the returned value and forwards it through the client-side chain in the reverse way. Eventually, the returned value is forwarded to the client application.

At deployment time, client and server side interceptors can be configured through a specific XML configuration file.

2.4. JBoss Instance Pooling Mechanism

As aforementioned, application servers maintain idle bean instances in an *instance pool*. These instances can be reused eliminating the necessity of creating a new one to process each incoming request that would surely be a time- and memory-consuming task.

JBoss' instance pool can be configured using an XML file. Configurable parameters include maximum size, minimum size and operation mode. Maximum/minimum size is the maximum/minimum number of instances the pool can store (default values are 100 and 0, respectively).

JBoss' pool can operate in two different modes: strict and non-strict. In the non-strict mode, instance pool can create any number of instances in order to process simultaneous requests, but it can keep and reuse only the configured maximum size. In the strict mode, instance pool can create only the configured maximum number of instances, no matter the number of simultaneous requests received.

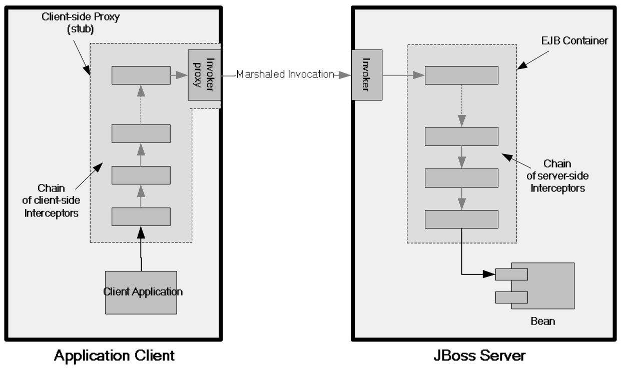

Figure 1. JBoss EJB architecture.

When the *instance interceptor* (see Section 2.3) receives an incoming request, it must require an available instance from the corresponding *instance pool*. At this point, if this pool has available instances, it returns one of them, otherwise, if the number of instances already created is lesser than the configured maximum size, the instance pool creates and returns a new one; else, the behaviour of the pool depends on its operation mode. If the instance pool is operating in non-strict mode, it creates and returns a new instance to the requestor. Otherwise, the incoming request must wait for an instance to become available. Additionally, a timeout can be configured to limit the maximum time that an incoming request can wait.

After obtaining an instance, the instance interceptor invokes the *next interceptor* in the chain. As soon as it receives the completion, it forwards the used bean instance back to the instance pool, which removes any information associated to the previous request. If the instance pool is not full, it can retain that instance. Otherwise, the instance becomes eligible to the garbage collection mechanism.

3. Modelling Approach

This section presents DSPN models for the instance pooling mechanism of the JBoss Application Server (or JBoss server, for short). In the first model, referred to as *base model*, it is assumed that the delays associated with timed transitions are exponentially distributed random variables with mean value obtained directly from measurement experiments. The second model proposed, referred to as *refined model*, is derived from the base one by replacing some selected timed transitions with *s-transition subnets* [15]. Both models have the same components: *Previous Interceptor*, *Instance Interceptor, Instance Pool* and *Next Interceptor*.

3.1. DSPN Base Model

The base model for the pooling mechanism is defined as $Pool=(P,T,I,O,H,G,M_0,\tau,W,\Pi)$. Figure 2 depicts this model, and Tables 1 and 2 contain information concerning G, τ, W and Π.

The *Previous Interceptor* acts as a workload generator representing clients' requests. A token in place *InterceptorReady* represents that this interceptor is ready to forward a new request. Transition *generateRequest* has an exponentially distributed firing time, modelling a Poison arrival process. Place *nrRequests* is just a counter used to get actual number of generated requests. Place *ReqGen* indicates that *Instance Interceptor* has a new request to process.

Instance Interceptor is responsible for obtaining a bean instance to be used in the request processing. It receives a request through the firing of its transition *invoke*. A token is generated in place *InvocationReceived* signalling that a bean instance must required from the *Instance Pool*.

Instance Pool is a repository of bean instances required through the transition *get*. After receiving a request, *Instance Pool* checks its operation mode, as indicated by place *CheckingOpMode1* and transitions *isStrict* and *isNotStrict* (which are mutually exclusive, as asserted by the enabling functions *#IsOpModeStrict=1* and *#IsOpModeStrict=0*, respectively).

In "non-strict mode", a token is generated in place *WaitingLog*, immediately enabling transition *log*, which represents the application server logging the request. In "strict mode", *Instance Pool* initially attempts to acquire a permission to get an instance and continue the request processing. If less than *MaximumSize* instances are in use, *attempt* transition fires, moving a token from place *WaitingAttempt* to place *WaitingLog*, enabling the transition *log*. On the other hand, if *MaximumSize* instances are actually processing simultaneous requests, current request waits and the corresponding token stays in place *WaitingAttempt*. Transition *timeout* is deterministic and

Table 1. Immediate transitions in the DSPN base model.

Transition	Weight (W)	Pri. (Π)	Enabling Function (G)
attempt	1	3	-
discard	1	1	#IsOpModeStrict=0
free	1	4	-
garbage	1	5	#Pool=MaximuSize
get	1	3	-
invoke	1	2	-
isNotStrict1	1	1	#IsOpModeStrict=0
isStrict	1	1	#IsOpModeStrict=1
poolIsEmpty	1	8	#Pool=0
poolIsNotEmpty	1	8	#Pool>0
release	1	4	#IsOpModeStrict=1

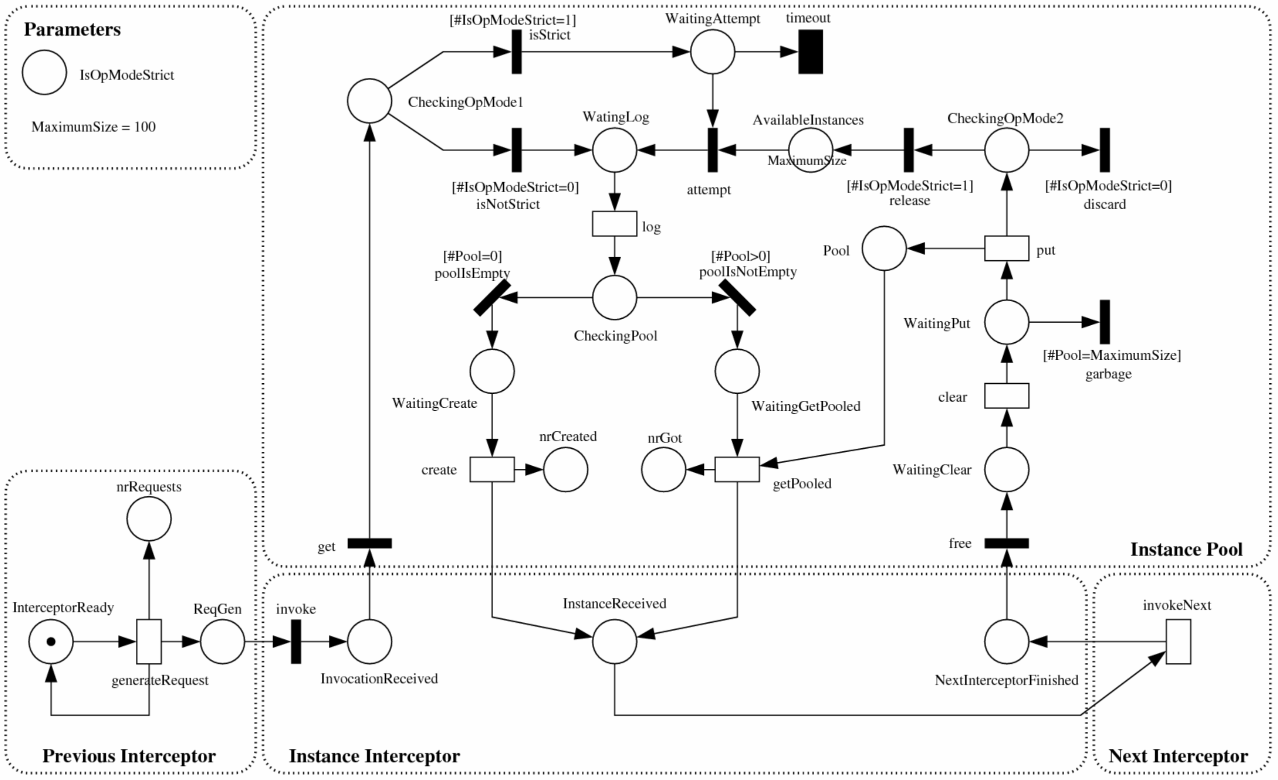

Figure 2. DSPN base model of pooling mechanism.

Table 2. Delays in the DSPN base model.

Transition (T)	Delay Time (τ) in μs
clear	*45.16*
create	*187.12*
generateRequest	*1021.74*
getPooled	*2.02*
invokeNext	*16.24*
log	*41.37*
put	*3.14*

fires when a request has to wait for the corresponding timeout period. If an instance is released before timeout, the waiting request will finally be processed.

After logging requests, *Instance Pool* checks place *Pool* that stores idle instances represented by tokens. This checking is indicated by place *CheckingPool* and transitions *poolIsEmpty* and *poolIsNotEmpty*. If *Instance Pool* already contains an idle instance (i.e. *#Pool* > 0), transition *poolIsNotEmpty* fires and generates a token in place *WaitingGetPooled*. This token enables transition *getPooled* that represents the activity of obtaining a pooled instance. Otherwise, if *InstancePool* has not an idle instance available (i.e. *#Pool* =0), transition *poolIsEmpty* fires and a token is moved to place *WaitingCreate*, meaning that a new instance must be created. In this model, instance creation is represented by *create* transition. A bean instance is returned to the *Instance Interceptor* in the form of a token deposited in place *InstanceReceived*.

After receiving a bean instance, *Instance Interceptor* associates it to the current request and invokes the next interceptor in the container chain, represented by *Next Interceptor* component in the DSPN model. The activity of the next interceptor has been modelled by the transition *invokeNext*. When this transition fires, the next Interceptor completes its execution and returns. This return is modelled by creating a token in place *NextInterceptorFinished*. After that, *Instance Interceptor* fires the transition *free,* releasing the used instance.

Upon been required to free an instance, *Instance Pool* disassociates it from any information related to the previous request (*clear* transition). Once the instance is cleared, a token is stored in place *WaitingPut* and the quantity of instances already stored in the pool is checked. If pool is full (*#Pool=MaximumSize*), transition *garbage* fires indicating that the used instance is now available to the garbage collection mechanism. Otherwise, transition *garbage* is disabled and transition *put* will eventually fire. When that occurs, a token is generated in place *Pool* indicating that the used instance can be reused. Transition *put* also places a token in place *CheckingOpMode2*. If *Instance Pool* is operating in "strict" mode, transition *release* fires signalling that another request can be processed. Otherwise, transition *discard* fires representing that the number of simultaneous requests is not controlled.

3.2. DSPN Refined Model

As mentioned before, an important assumption of the base model (see Figure 2) refers to the delays associated to the timed transitions, which are exponentially distributed random variables with mean value equals to the mean of the measured delays. However, the measured values do not seem to follow an exponential distribution, as can be inferred by their mean and standard deviation presented in Table 3. In exponential distributions, mean (μ) and standard deviation (σ) tend to have the same value leading to an unitary coefficient of variation. In effect, other strategies, like Chi-square and Kolmogorov, can be used to evaluate the quality of this approximation.

In general, a good way of dealing with generally distributed random variables is to represent them using a combination of exponential ones in a way that some moments of the general distribution match corresponding moments of the exponential composition. These combinations of exponential distributions are known as Phase Type distributions (PH distributions) [12]. Algorithms that do this kind of mapping, from a general distribution to a PH distribution, are called *moment matching algorithms.* In fact, a moment matching has already been done in the base model when an empirically distributed variable was approximated by an exponential one promoting a matching between the first moments of both distributions (i.e., their mean values). Better results, however, can usually be obtained by using algorithms that match other moments besides the first one.

A refined model is derived from the base one using the moment matching algorithm proposed by Watson III in [15]. This algorithm takes advantage of the fact that Erlangian distributions usually have μ ≥ σ, while Hyperexponential distributions generally have μ ≤ σ, to propose the representation of an activity with a generally distributed delay as an Erlangian or a Hyperexponential subnet referred to as s-transition. Therefore, according to

Table 3. Mean (μ), std. deviation ( ) and coef. of variation for each timed task.

Transition	μ (μs)	σ (μs)	Coef. of Variation
clear	45.16	364.56	8.07
create	187.12	477.77	2.55
generateRequest	1,021.74	4,572.40	4.48
getPooled	2.02	6.45	3.20
invokeNext	16.24	299.12	18.46
log	41.37	323.14	7.81
put	3.14	59.66	19.01

the coefficient of variation (σ/μ) associated to the delay of an activity, an appropriate s-transition implementation can be selected. For each s-transition implementation, there are parameters that can be configured in such way that the first and second moments associated to the delay of the original activity match with the first and second moments of s-transition as a whole.

According to the aforementioned algorithm, all timed transitions presented on the base model should be approximated by hyperexponential s-transitions, once their corresponding coefficients of variation are greater than one (see Table 3). The hyperexponential implementation for an s-transition proposed by Watson III is presented in Figure 4. To approximate a generally distributed transition having mean equals to μ and standard-deviation equals to σ, the parameters of the hyperexponetial subnet are configured to $r_1 = 2\mu^2/(\mu^2+\sigma^2)$, $r_2 = 1 - r_1$ and $\lambda = 2\mu/(\mu^2+\sigma^2)$. Parameter values associated to the

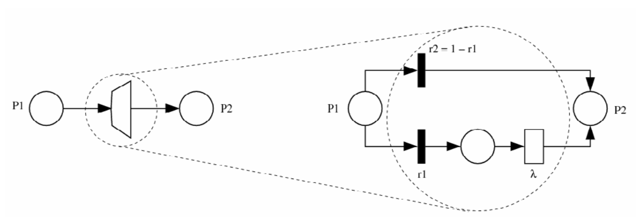

Figure 4. Hyperexponential implementation for an s-transition.

s-transitions used in the refined model are presented on Table 4.

Figure 3 depicts a Petri net that comprehends the core of the refined model. This net is referred to as *control net* once it has complete control of the dynamics of the system modelled. It is worth observing that all timed transitions approximated by s-transition subnets are carefully removed from the control net and represented in

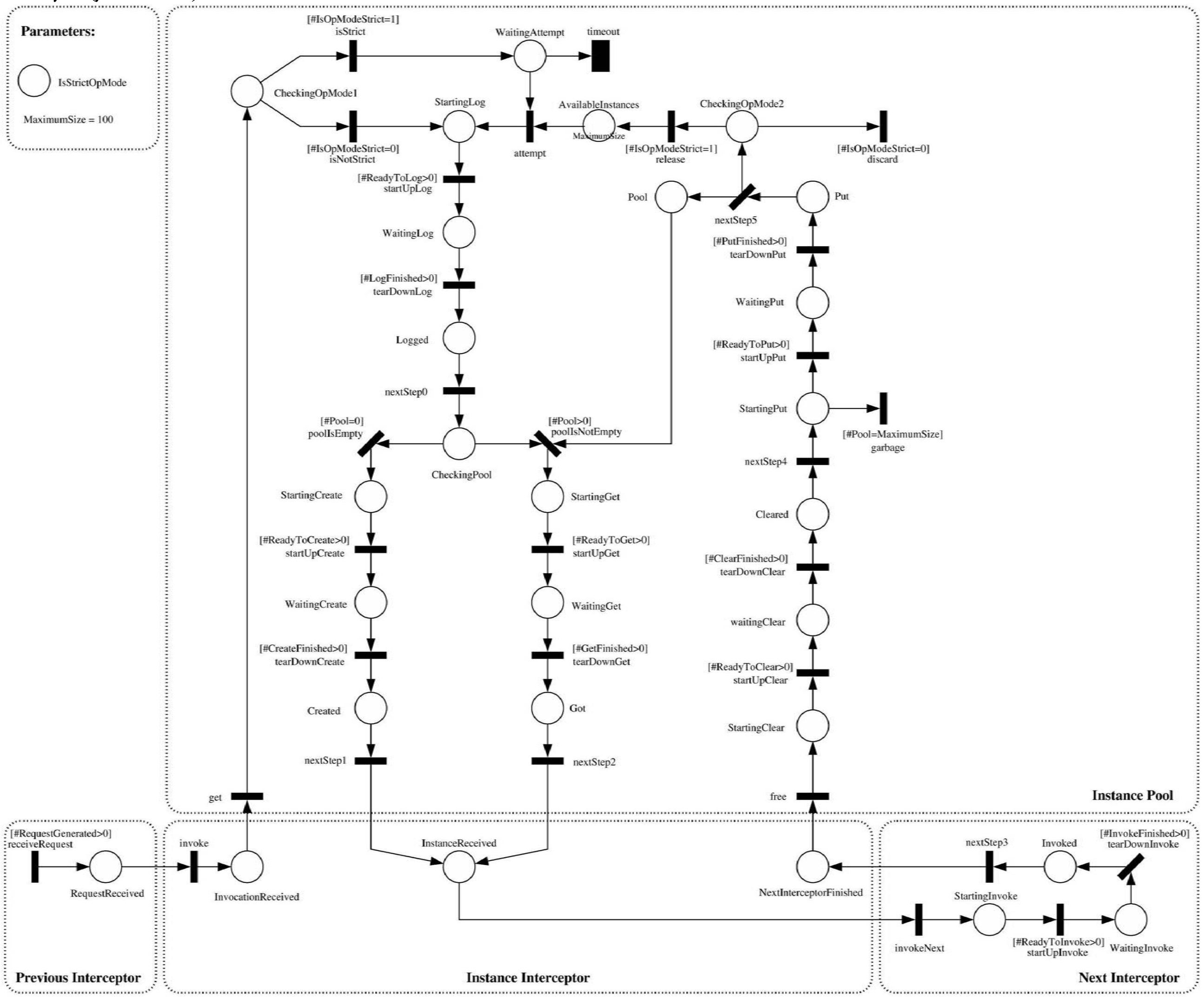

Figure 3. DSPN refined model: control net.

Table 4. Parameters for the s-transitions used in the refined model.

Transition	r_1	r_2	λ
clear	0.0302	0.9698	0.0007
create	0.2660	0.7340	0.0014
generateRequest	0.0951	0.9049	0.0001
getPooled	0.1786	0.8214	0.0884
invokeNext	0.0059	0.9941	0.0004
log	0.0323	0.9677	0.0008
put	0.0055	0.9945	0.0018

corresponding subnets (see Figure 5). Control net and s-transition subnets are connected using a set of places and transitions. The utilization of this set of places and transitions (referred to as *connectors*) associated with a carefully chosen set of enabling functions and priorities allows the substitution of each selected timed transition by the corresponding s-transition subnet without significantly changing the overall model.

4. Validation of the DSPN Models

The benchmark application used to calibrate and validate the proposed models comprises a stateless session bean and some web components. Clients access this application sending HTTP requests that are processed by the web components. These components invoke a bean instance that returns a constant string, which is placed in an HTML page and sent back to the clients.

Ten heavy demanding virtual clients simultaneously access the benchmark application, each one trying to perform 100 req/s. Each experiment consists in running those clients during 100 seconds.

Tests were realized in an isolated Fast-Ethernet network containing only a client and a server machine. The client machine is an Athlon 2000+ with 768MB of RAM running Windows 2000 Professional and is used only to simulate client requests. The server machine is a Pentium M 1.6MHz with 768MB of RAM running JBoss application server version 3.2.7. Java HotSpot Client Virtual Machine (version 1.5.0) was used to run JBoss. Heap was configured with an initial size of 256Mb and a maximum size of 512Mb. Additional applications and services in each machine were stopped in order to minimize external interference. Before starting each experiment, it was realized a warm-up composed by one client executing 10000 requests in a row.

4.1. Performance Results and Analysis

To validate the proposed models, the benchmark application was set to use the default instance pool configuration having a minimal size of 0 and maximum size of 100 and operating in non-strict mode. Then, the described workload was submitted and the number of bean instances created was logged. The same workload was considered in simulation experiments performed on both base and refined models.

Figure 6 shows the number of contexts created in both, measurements (continuous line) and simulations carried

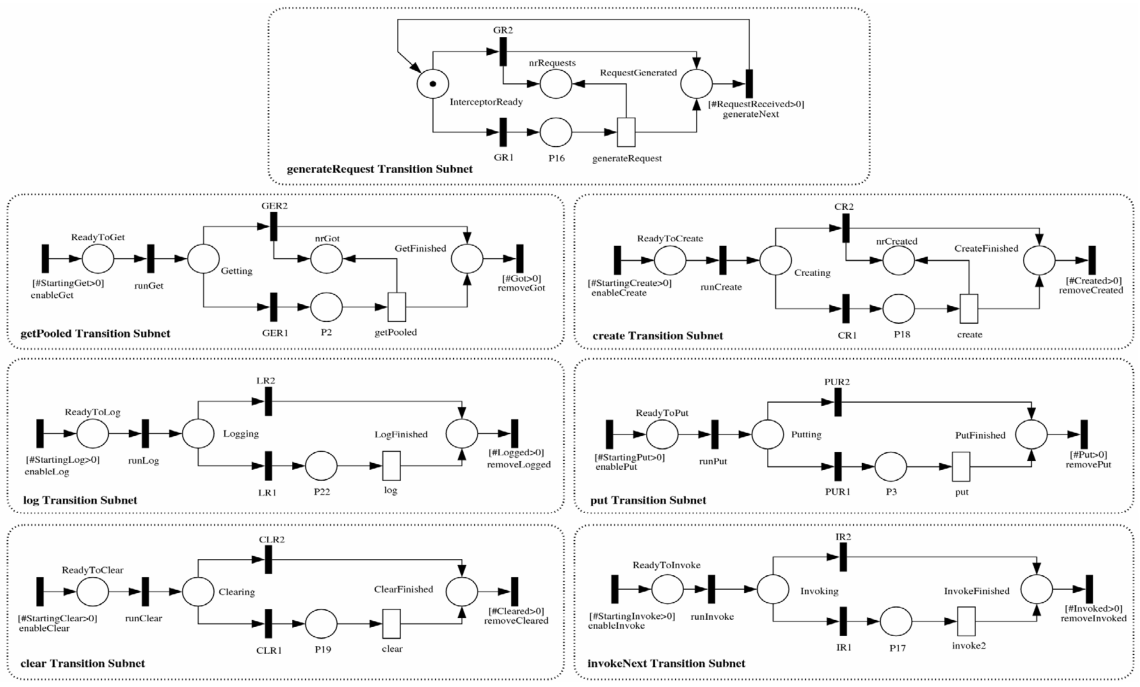

Figure 5. DSPN refined model: s-transitions subnets.

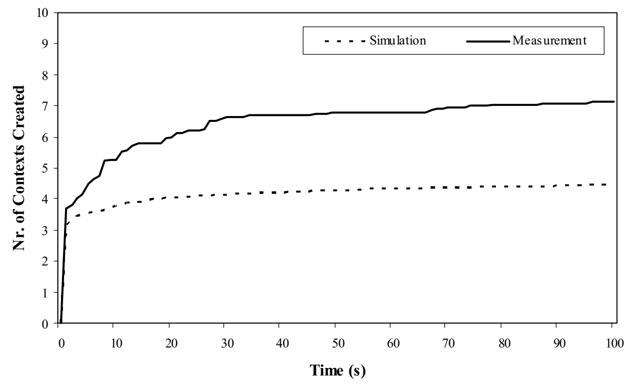

Figure 6. Context creation for base model.

out using the base model (dashed line). Figure 7 presents a comparison involving measurement (continuous line) and simulations using the refined model (dashed line).

Those results demonstrate that the DSPN refined model approximates closely the actual JBoss pooling mechanism.

5. Conclusions

This paper has investigated the utilization of simulation models for performance prediction of application servers. In order to do this, a DSPN model that represents the pooling mechanism used by JBoss application server was designed. It is worth observing that the pooling mechanism has an important impact on the overall performance of an application server.

The simulation results obtained using the DSPN refined model approximate measurement results closely, validating both, the approach and the model itself. This point is considered the main contribution of this work.

There are some open questions concerning the development of a complete performance model. In particular, the replication of this approach in the identification and modelling of other relevant services concerning application servers is been considered. Additionally, it is currently being developed a Petri net model for a workload generator in order to allow the simulation a variety of scenarios.

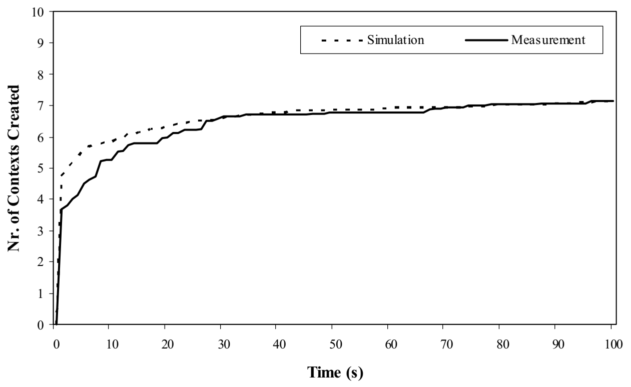

Figure 7. Context creation for refined model.

References

[1] Cecchet, E., Marguerite, J., and Zwaenepoel W., “Performance and scalability of EJB applications”. In Proc. 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2002.

[2] CSIRO Middleware Technology Evaluation Series: Evaluating J2EE Application Servers version 2.1, 2002.

[3] Fleury, M., Reverbel, F., “The JBoss Extensible Server”, Proc. International Middleware Conference, 2003.

[4] Jain, R., “The Art of Computer Systems Performance Evaluation”, Wiley Computer Publishing, 1991.

[5] Kounev, S. and Buchmann, A., “Performance Modeling and Evaluation of Large-Scale J2EE Applications”. In Proceedings of the 29th Int. Conf. of the Computer Measurement Group on Resource Management and Perf. Evaluation of Enterprise Computing Systems, 2003.

[6] Kounev, S., Weis, B. and Buchmann, A., “Performance Tuning and Optimization of J2EE Applications on the JBoss Platform”. In Journal of Computer Resource Management, Issue 113, 2004.

[7] Liu, Y., Fekete, A., Gorton, Y., “Predicting the Performance of Middlewarebased Applications at the Design Level”, In Proc. 4th Int. Workshop on Soft. and Performance, 2004.

[8] LLadó, C. M., Harrison, P. G.. “Performance Evaluation of an Enterprise JavaBeans Server Implementation”. In Proc. 2nd Int. Workshop Soft. and Performance, 2000.

[9] Marsan, M., Balbo, G., Conte, G., “A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems”, ACM Transactions on Computer Systems, vol. 2, pages 93-122, 1984.

[10] Marsan, M., Chiola, G., “On Petri Nets with Deterministic and Exponentially Distributed Firing Times”, LNCS 266, pages 132-145, Springer-Verlag, 1987.

[11] Mc Guinness, D., Murphy, L., “A simulation model of a multi-server EJB system”, A-MOST’05, 2005.

[12] Neuts ,M. F., “Probability distributions of phase type.”, In: Liber Amicorum Professor Emeritus H. Florin, University of Louvain, Belgium, pages 173-206, 1975.

[13] Transaction Processing Performance Council, “TPC Benchmark App (Application Server)”, v1.1.1, 2005.

[14] Vinoski, S., "The Performance Presumption", IEEE Internet Computing, vol. 07, No. 2, 2003.

[15] Watson III, J., Desrochers, A., “Applying Generalized Stochastic Petri Nets to Manufacturing Systems Containing Nonexponential Transition Functions”, IEEE Transactions on Systems, MAN, and Cybernetics, vol. 21, No. 5, 1991.