Using Stochastic Petri Nets for Performance Modelling of JBoss Application Server

Fábio N. Souza¹, Roberto D. Arteiro¹, Nelson S. Rosa¹, Paulo R. M. Maciel¹

¹Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 15.064 – 91.501-970 – Recife – PE – Brazil
{fns,rda,nsr,prmm}@cin.ufpe.br

Abstract. J2EE application servers have been widely adopted as distributed infrastructure (or middleware) for developing distributed systems. The current approaches for performance evaluation of application servers have mainly concentrated on the adoption of measurement techniques. This paper focuses on the use of simulation techniques and presents stochastic Petri net models for performance evaluation of the JBoss application server. In order to validate the proposed models, simulation results are compared with the ones that have been measured using an actual JBoss implementation.

1. Introduction

Java 2 Platform Enterprise Edition (J2EE) Specification defines an environment for supporting the development and deployment of distributed enterprise applications. Numerous product vendors offer implementations of this specification which are referred to as J2EE application servers. This crowded marketplace creates a new challenge for IT departments which involves the evaluation and selection of an adequate J2EE product. Choosing an implementation is a hard task since 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 all their qualities is too hard [Vinoski 2003].

Three different techniques can be used in performance evaluations: measurement, analytical modelling and simulation. The measurement approach has been widely used in the performance evaluation of application servers. [Cecchet et al. 2002] 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) [CSIRO 2002] provides a detailed evaluation and comparison of different application servers, presenting a qualitative analysis of their features and a quantitative analysis based on performance and stress tests performed using a stockbroking benchmark application. [Kounev et al. 2004] 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) [TPC 2005] specifies TPC Benchmark™ App (TPC-App), which is an application server and web services benchmark. Despite of its utilization, the measurement approach has important limitations. Firstly, its high sensibility to variations in environment parameters may 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.

Analytical models can derive performance results quickly, providing valuable performance information without having extra costs associated with the replication of the real environment. [Lladó 2000] presents an analytical model, based on queuing theory, for an idealized application server. Using that model important trends can be inferred. However, the performance results obtained are not validated against any real server. [Kounev and Buchmann 2003] uses Non-Product-Form Queuing Networks to propose an analytical model for a cluster of application servers running the SPECjAppServer2002 J2EE benchmark application. In spite of the impressive results obtained for CPU and throughput, the proposed model can not predict response times accurately because software contention is not represented. [Liu et al. 2004] 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. This research is particularly promising as it can anticipate performance before the system is completely developed. A point to comment, however, is that some important metrics like those related to resources utilization can not be derived. Despite those important research mentioned, analytical modelling, in general, requires so many simplifications and assumptions that imposes difficulty to obtain accurate results [Jain 1991]. Another problem concerning some analytical modelling tools is known as the state-space 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. [McGuinness and Murphy 2005] 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. Since the proposed model has many configuring parameters, the corresponding flexibility is high. An important point to mention, however, is the lack of validation against a real server. Despite of their strengths, simulation models generally present limitations. In fact, complex simulation models usually require much 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 present an approach to performance prediction of application servers using stochastic Petri nets. Petri net models of the JBoss application server are proposed and simulation is used to evaluate performance metrics in some considered scenarios. The results obtained are compared with measurements performed in a controlled environment. Particularly, we focus on the analysis of such application server due to its enormous utilization in the marketplace, its open source characteristic, and its EJB compliance. In turn, the adoption of Petri Nets has been based on four basic facts: it is a suitable formalism for system performance evaluation; it enables us to build models that may be both simulated and analyzed; it allows a natural modelling of various services offered by application servers (e.g. instance pool, control of concurrency); and there is a large number of tools available.
The remainder of this paper is organized as follows. Theory of Stochastic Petri Nets is briefly presented in Section 2. Section 3 introduces the basic concepts of J2EE and application servers. Section 4 presents the Petri net models designed. Next, Section 5 presents the validation of the developed models using simulation and measurement techniques. Finally, concluding remarks and future works are presented in Section 6.

2. Stochastic Petri Nets

Petri nets are a family of formal specification techniques that allows for a graphical, mathematical representation and have powerful methods, which allow 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) [Marsan et. al 1984] is one of the most extensively adopted classes of stochastic Petri nets and can be defined as a tuple \((P, T, I, O, H, G, M0, W, \Pi)\) 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 whether the transition can be enabled;
- \(M0\) 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 (which is a probabilistic factor used in the selection of the next transition that will fire when more than one of them is concurrently enabled); and
- \(\Pi\) associates to each immediate transition a natural number that represents the respective transition priority level. An immediate transition can only be enabled if there is no other one with a higher priority level currently enabled.

The set of places represents the set of resources, local states and system variables. The set of transitions represents the set of 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. Transitions are fired under interleaving firing semantics, a common semantics adopted even in the untimed place/transition model. However, immediate transitions have priority higher than those timed ones. Besides that, each timed transition has an associated firing semantic that can be single-server, infinite-server or multiple-server (or k-server). In single-server semantic, an activity executes sequentially, whereas in the infinite-server semantic, an unlimited set of the servers processes, in parallel, many instances of the same activity. Finally, the multiple-server semantic limits the number of available servers to \(k\).

3. J2EE and JBoss Application Server

3.1. J2EE and Enterprise Java Beans

Java 2 Platform, Enterprise Edition specification [Sun 2003b] defines a standard environment to support the development and deployment of Java distributed applications. Implementations of this specification are referred to as J2EE application servers.

The core of J2EE specification is the definition of a framework for the development of server-side components known as Enterprise JavaBeans (EJBs) [Sun
Instances of EJB components (bean instances) are hosted in a runtime environment, named EJB container, which takes responsibility of managing their life cycle, managing resources in their behalf and providing them with predefined system-level services such as transaction management and security. EJBs call on these services declaratively by configuring them in a XML file known as deployment descriptor. In order to provide those services transparently, an EJB container acts as a proxy interposing itself between clients and bean instances.

According to EJB 2.1 specification [Sun 2003a], there are three types of EJB components: Session beans, Entity beans and Message-Driven beans. Session beans are non-persistent components commonly used to represent business processes. These components are not sharable, meaning that at any moment in time, each session bean instance represents a single client executing inside the application server. Session beans are classified as statefull or stateless, depending on if they retain or not information between consecutive requests. Entity beans are sharable and persistent components used to represent business data. Entity beans may be classified as CMPs or BMPs according to who is responsible for persisting their state: the container or the bean itself, respectively. Finally, message-driven beans are used to enable J2EE application to process requests asynchronously. They are implicitly invoked by sending messages to a queue associated with it.

3.2. EJB in JBoss

In the JBoss architecture [Fleury and Reverbel 2003][JBoss 2004], a client application (local or remote) has access to an EJB component through a client-side proxy which exposes the same methods that are exposed by the component itself. Each client-side proxy has a chain of interceptors. When a client application invokes one of the proxy’s methods, this proxy collects information about the invoked method in an object named invocation and delivers this 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 this information to the invocation object and forwards this object to the next interceptor in the chain. Its last interceptor is the invoker interceptor, which is in charge of forwarding the invocation object that represents the request to a server side component known as invoker. If client and server are running on the same virtual machine the invocation object is simply passed by reference, otherwise it is marshaled and transferred through the network to the server side.

At the server side, when an EJB component is deployed in JBoss application server a corresponding EJB container is created. This container is, then, associated with a server-side component named invoker, whose role is to receive invocations from invoker proxies, unmarshal them if necessary and forward them to corresponding containers.

As any compliant EJB container, that one offered by JBoss is responsible for managing the bean instances and providing them with predefined services. However, instead of implementing these services itself, a JBoss container has a chain of pluggable server-side interceptors, each responsible for implementing a specific service. 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 carry out the corresponding service and, then, forwards the invocation to the next one down in the chain. Important interceptors usually present in server’s chain are the security, the transaction and the instance interceptors. The security interceptor gets the information concerning the user that is invoking the bean and checks for authorization. The transaction interceptor takes care of transactional properties, including distribution aspects. The instance interceptor is responsible for obtaining a bean instance to be used in the request processing (Section 3.3 details JBoss instance pooling mechanism). Finally, the last interceptor in server’s chain is responsible for identifying the method required, and invoking it using the obtained instance. The return of the method transverses the chain in reverse order until been received by the container which deliveries it back to the invoker. The invoker then marshals the returned value (if necessary) and sends it back to the invoker proxy. This invoker proxy unmarshalls (again, if necessary) the returned value and forwards it through the client-side proxy’s chain in the reverse way. After that, returned value is finally forwarded to the client application.

![Diagram of JBoss EJB Architecture](image)

**Figure 1. JBoss EJB Architecture.**

During deployment time, client and server side interceptors can be selected and configured through JBoss specific XML configuration files. Figure 1 shows a client-side proxy and an EJB container interacting in order to fulfill user requests.

### 3.3. JBoss Instance Pooling Mechanism

An important service that can be provided by an EJB container is Instance Pooling. When an EJB container intercepts a client’ request it may create a new bean instance to process this request. Otherwise, if there is an instance already created in memory it may be better to reuse it so reducing both memory and delay necessary to process the corresponding request. Sometimes, an EJB container may also reduce allocated resources by destroying bean instances that have not been used for a long time. All this instance management is named Instance Pooling. It is important to observe that the exact mechanism used in instance pooling is not part of EJB specification, so it is EJB container specific. In this section, we investigate JBoss default implementation of this important mechanism and its configuration.
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 that the pool can store (default values are 100 and 0, respectively). By the time the first client’s request comes in, the instance pool creates and initializes this minimum number of instances configured.

JBoss pool can operate in two different modes: strict and non-strict. In 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 how many simultaneous requests it receives.

When an instance interceptor (see Section 3.2) receives an incoming request, it must require an available instance from the corresponding instance pool. If this pool has available instances, it returns one of them; otherwise, if the number of instances already created is less than pool’s maximum size, the instance pool creates and returns a new one; else, pool’s behaviour depends on its operation mode. If it 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 in the pool. Additionally, to define the maximum time that an incoming request must wait, a timeout can be configured.

After obtaining an instance, the instance interceptor invokes the next interceptor in the container’s chain. Once it receives the completion, it forwards the used bean instance back to the instance pool, which deletes any information associated to the previous request. If the instance pool is not full, it can retain that instance; otherwise, the instance will become eligible by the garbage collection mechanism. Figure 2 depicts the aforementioned interactions between instance pool, instance interceptor and its previous and next interceptors in server’s chain.

4. GSPN Models
This section presents GSPN models for JBoss application server (or JBoss server, for short). The proposed models represent a scenario composed of a set of clients carrying...
out HTTP requests following a Poisson distribution\(^1\) and a JBoss server hosting a J2EE application. This application is a benchmark application developed specifically to be used in our experiments and is composed of a single stateless session bean and some web components. Clients access this application by sending HTTP requests that are processed by the web components. These components invoke a bean instance that returns a constant string. The information returned is placed in an HTML page that is sent back to the client.

The first designed model is referred to as base model. In this model, it is assumed that delays associated with timed transitions are exponentially distributed random variables with mean value obtained directly from measurement experiments.

The second proposed model, referred to as refined model, is derived from the base one replacing selected timed transitions with \textit{s-transition subnets} [Watson III and Desrochers 1991] having mean delay time and standard deviation matching the measured values.

Both models have the same three components: clients, network and JBoss application server. It is also important to mention that both models depict the instance pool operating in non-strict mode, so that the number of bean instances created is not constrained.

### 4.1. GSPN Base Model

Base model is depicted in Figure 3. In this model, clients are represented by the subnet comprising places ClientsReady and SendingRequest, transition generateRequest and its incoming and outgoing arcs. In its initial marking, ClientsReady has \( nrClients \) tokens, each one corresponding to an active client. In order to model simultaneous clients with request rates following a Poisson distribution, transition generateRequest is exponential and has infinite server semantics. Once a request is generated, a token is put in place SendingRequest representing that this request is now being sent over the network. The delay of generateRequest transition (\( dRequest \)) is a parameter that can be varied in order to represent different request rates.

Network component is approximated by a subnet comprising two timed transitions with exponential delays, sendRequest and sendResponse. Corresponding delay parameters (latency) can be approximated based on the size of the information that is exchanged between client and server.

JBoss server is itself modeled as a composition of four distinct components: PreviousInterceptors, InstanceInterceptor, InstancePool and NextInterceptor. These components share access to server’s CPUs, which are represented by tokens in place CPUs. Before performing any time consuming task, a component must allocate an available CPU. This allocation is carried out in the proposed model by those immediate transitions whose names start with sync. After acquiring a CPU, a component enters in one of its local states represented by places whose names start with Waiting and remains in this state until the corresponding task is finished and the allocated CPU is released.

---

\(^1\) According to [Jain 1991], a Poisson distribution is particularly appropriate whenever the requests arrival are from a large number of independent sources. Such arrival processes are called Poisson processes and are generated by a negative exponential distribution.
The *PreviousInterceptors* component represents the JBoss elements that are responsible for receiving incoming requests and forwarding outgoing responses. In JBoss, incoming requests cross two chains of interceptors, one contained by the proxy and another by the server. In the proposed model, incoming requests are represented by tokens in place *RequestsReceived* and the work carried out by all the incoming interceptors before the instance one is invoked is represented by a single transition, *invokeInChain*. In its turn, bean’s response also crosses these chains, but in the reverse way. Again, all the work carried out by the interceptors after the instance one returns is represented by a single timed transition, *outChain*. When this transition fires, representing the conclusion of the interceptors’ tasks, a token is created in place *SendingResponse* representing that the outgoing response is now ready to be forwarded back to the client.

*InvokerInterceptor* component is responsible for obtaining a bean instance to be used in the request processing. It receives requests as tokens in place *InvocationReceived*. To process the incoming request, the *InstanceInterceptor* requests a bean instance from *InstancePool* component.

*InstancePool* component acts as a repository of bean instances. After receiving a request, *InstancePool* logs this fact in JBoss log file, what is represented by transition *log*. After logging requests, *InstancePool* checks place *Pool* that stores idle instances represented by tokens. This checking is indicated by the place *CheckingPool* and transitions *poolIsEmpty* and *poolIsNotEmpty*. If pool already contains an idle instance (i.e. *Pool* > 0), *poolIsNotEmpty* transition fires and generates a token in *WaitingGet* place. This token enables *getPooled* transition that represents the activity of obtaining a pooled instance. Otherwise, if pool has not an idle instance available (i.e. *Pool* =0), transition *poolIsEmpty* fires and a token is moved to place *WaitingCreate*. This means...
that a new instance must be created. In this model, instance creation is represented by	transition \textit{create}. Each instance created is counted by putting a token in place \textit{NrCreated}. In order to turn the Petri net structurally limited, place \textit{CreateLimit} and transition \textit{T1} are also represented. Place \textit{CreateLimit} is initialized to \textit{nrClients} in order to guarantee that, if necessary, each client can create an instance in order to process its request, however it will usually reuse an instance already created. Transition \textit{T1} has a huge delay, so that it does not affect the instance creation counter. A bean instance is returned to \textit{InstanceInterceptor} in the form of a token deposited in place \textit{InstanceReceived} (whether it has been created or obtained).

After obtaining an available instance, \textit{InstanceInterceptor} calls the \textit{NextInterceptor} component. Activities carried out by all the next interceptors down in server’s chain and by the bean processing itself are represented together as a unique timed transition: \textit{invokeNext}. When this transition fires, the used bean instance is returned to the \textit{InstanceInterceptor} as a token created in place \textit{NextInterceptorFinished}. Finally, the \textit{InstanceInterceptor} can return the used instance to the \textit{InstancePool}.

When the \textit{InstancePool} receives a bean instance, it clears all the information associated with the concluded request processing. This task is modeled by the timed transition \textit{clear}. After that, pool will check its current and maximum size in order to decide if that instance should be discarded (become eligible for garbage collection) or be retained for further reuse. This decision is modeled by places \textit{ReleasingBean} and \textit{FreeSlots}, and transitions \textit{discard} and \textit{goToPut}. After the bean has been marked as eligible for garbage collection or stored in pool’s slots, the response crosses the chain of previous interceptors which are represented together by timed transition \textit{outChain}. After that, the response will finally be forwarded to the client, this is represented by the \textit{sendResponse} transition.

To complete the model description, it is necessary to adjust the delay of the modeled transitions. This step is referred to as parameterization and involves carrying out a series of experiments conducted to measure the delays associated with the activities represented by the transitions. Each performed experiment consists of a unique client executing 50,000 requests in a row. The first 10,000 of these are intended to warm-up the JBoss server and the corresponding JVM. In order to minimize interference between two consecutive experiments, the JBoss server is restarted at the end of each one. By the end of the parameterization, mean and standard-deviation are determined for each measured delay, considering the last 40,000 requests. The data collected in those experiments are summarized in Table 1.

\begin{table}[h]
\centering
\caption{Mean ($\mu$), Standard Deviation ($\sigma$) for base models timed activities.}
\begin{tabular}{|l|c|c|}
\hline
Transition  & $\mu$ ($\mu$s) & $\sigma$ ($\mu$s) \\
\hline
 clear & 15.495 & 21.813 \\
 getPooled & 1.967 & 0.987 \\
 invokeNext & 24.812 & 35.498 \\
 log & 7.976 & 43.629 \\
 put & 1.977 & 0.436 \\
\hline
\end{tabular}
\end{table}
The main details concerning the timed transitions are summarized in Table 2. The delays of the sendRequest and sendResponse are calculated using the size of the request and response packets exchanged between clients and server, and the network bandwidth, while those associated with invokeInChain and outChain are estimated. The other delays represent the mean of the measured values shown in Table 1. Finally, the details concerning the immediate transitions are presented in Table 3.

### Table 2. Timed transitions in the GSPN base model.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Delay Time in μs</th>
<th>Firing Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>15.495</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>create</td>
<td>3504.822</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>getPooled</td>
<td>1.967</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>invokeInChain</td>
<td>0.500</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>invokeNext</td>
<td>24.812</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>log</td>
<td>7.976</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>outChain</td>
<td>0.500</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>put</td>
<td>1.977</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>sendRequest</td>
<td>21.057</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>sendResponse</td>
<td>48.752</td>
<td>Infinite-Server</td>
</tr>
</tbody>
</table>

### Table 3. Immediate transitions in the GSPN base model.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Weight</th>
<th>Priority</th>
<th>Enabling Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>discard</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>poolIsEmpty</td>
<td>1</td>
<td>1</td>
<td>#Pool=0</td>
</tr>
<tr>
<td>poolIsNotEmpty</td>
<td>1</td>
<td>1</td>
<td>#Pool&gt;0</td>
</tr>
<tr>
<td>syncClear</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>syncGet</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>syncInvokeInChain</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>syncInvokeNext</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>syncLog</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>syncOutChain</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>syncPut</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

### 4.2. GSPN Refined Model

As mentioned before, one important assumption implied in base model (see Figure 3) is that the transitions’ delays are exponentially distributed random variables with parameter equals to the mean of the measured delays. However, the measured values do not follow an exponential distribution, as can be seen by their mean and standard deviation presented in Table 1. In exponential distributions, mean (\(\mu\)) and standard deviation (\(\sigma\)) have the same value leading to a coefficient of variation (\(\sigma/\mu\)) with a unitary value. In effect, strategies that are more sophisticated can be used to evaluate the quality of this approximation like Chi-square and Kolmogorov.

In general, a good way to deal with general distribution random variables (like those associated with the delays measured) is to represent those distributions 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) [Neuts 1975]. Algorithms that do this kind of mapping from a general distribution to a PH distribution are called moment matching algorithms. In fact, we have already done a moment matching in our base model when we approximated an empirically distributed variable by an exponential one promoting a matching between the first moments of both distributions (i.e., its mean value). Better results, however, can be usually obtained using other algorithms that match other moments besides the first one.

Figure 4. A hyperexponential implementation for an s-transition.

To refine the base model we have used a moment matching algorithm proposed in [Watson and Desrochers 1991]. This algorithm takes advantage of the fact that Erlangian distributions usually have $\mu \geq \sigma$, while Hyperexponential distributions generally have $\mu \leq \sigma$, 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 the coefficient of variation associated an activity’s delay, an appropriate s-transition implementation model can be selected. For each s-transition implementation model, 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.

Table 4. Coefficient of Variation for base models timed activities approximated by s-transition subnets.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Coefficient of Variation</th>
<th>s-Transition Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>1.4078</td>
<td>Hyperexponential</td>
</tr>
<tr>
<td>getPooled</td>
<td>0.5018</td>
<td>Erlangian</td>
</tr>
<tr>
<td>invokeNext</td>
<td>1.4367</td>
<td>Hyperexponential</td>
</tr>
<tr>
<td>log</td>
<td>5.4703</td>
<td>Hyperexponential</td>
</tr>
<tr>
<td>put</td>
<td>0.2204</td>
<td>Erlangian</td>
</tr>
</tbody>
</table>

According to the aforementioned algorithm, transitions clear, log and process should be approximated by hyperexponential s-transitions, since corresponding coefficients of variation are greater than one (Table 4). The hyperexponential implementation for an s-transition proposed by Watson III is presented in Figure 4. To approximate one of such timed transitions having mean equals to $\mu$ and standard-deviation equals to $\sigma$, we configure hyperexponential’s parameters to $r_1 = 2\mu^2/(\mu^2 + \sigma^2)$, $r_2 = 1 - r_1$ and $\lambda = 2\mu/(\mu^2 + \sigma^2)$ so that the first and second moments can match. The values for those three parameters associated with corresponding timed transitions used in the refined model are presented in Table 5.
Table 5. Parameter values for Hyperexponential s-transition implementations used in refined model.

<table>
<thead>
<tr>
<th>Transition</th>
<th>r1</th>
<th>r2</th>
<th>λ</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>0.6707</td>
<td>0.3293</td>
<td>0.0432881485</td>
</tr>
<tr>
<td>invoke</td>
<td>0.6564</td>
<td>0.3436</td>
<td>0.0264557755</td>
</tr>
<tr>
<td>log</td>
<td>0.0647</td>
<td>0.9353</td>
<td>0.0081089375</td>
</tr>
</tbody>
</table>

Table 6. Parameter values for Erlangian s-transition implementations used in refined model.

<table>
<thead>
<tr>
<th>Transition</th>
<th>γ</th>
<th>λ₁</th>
<th>λ₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPooled</td>
<td>3</td>
<td>2.3815</td>
<td>1.9387</td>
</tr>
<tr>
<td>put</td>
<td>20</td>
<td>28.7280</td>
<td>10.2998</td>
</tr>
</tbody>
</table>

Transitions get and put have coefficients of variation lesser than one (Table 4) and should be mapped to Erlangian s-transitions as depicted in Figure 5. Erlangian s-transitions have three parameters: λ₁ and λ₂ that are the rates of the exponential elements; and γ that is an integer representing the number of phases of this distribution. These parameters can be calculated as: \( \frac{\mu}{\sigma^2} - 1 \leq \gamma \left(\frac{\mu}{\sigma^2}\right) \), \( \lambda_1 = \frac{1}{\mu_1} \) and \( \lambda_2 = \frac{1}{\mu_2} \), where:

\[
\mu_1 = \frac{\mu + \sqrt{\gamma(\gamma + 1)\sigma^2 - \gamma \mu^2}}{\gamma + 1}, \quad \mu_2 = \frac{\gamma \mu + \sqrt{\gamma(\gamma + 1)\sigma^2 - \gamma \mu^2}}{\gamma + 1}.
\]

The values of these parameters for transitions get and put are presented in Table 6.

Figure 5. An Erlangian implementation for an s-transition.

Figure 6 depicts a Petri net that comprehends the core of the refined model. We call this the control net once it has complete control of the dynamics of the system modeled. It is worth observing that all timed transitions approximated by s-transition subnets (those indicated in Table 4) are carefully removed from the control net and represented in corresponding subnets (see Figure 7). Control net and s-transition subnets are connected using a set of places and transitions. This set comprises all places and transitions whose names start with Starting, Ready, startUp, tearDown, enable, run and remove. The utilization of this set of places and transitions (that we refer to as connectors) associated with a carefully chosen set of enabling functions and priorities allows us to substitute each selected timed transition by the corresponding s-transition subnet.
5. Validation of the GSPN Models

Four workload models were used in order to validate the proposed Petri net models. The first three comprise 5, 10 and 15 heavy demanding virtual clients simultaneously
accessing the benchmark application. Each client tries to perform one hundred requests per second. In the forth workload model 100 users are simulated, each one generating one request per second. Various experiments were carried out for each workload model, each one consisting of running virtual clients for 100 seconds. Virtual clients were simulated using JMeter 2.0.3, which is a Java-based tool for load testing client-server applications.

Tests were carried out in an isolated Fast-Ethernet network containing just a client and a server machine. Client machine is an Athlon 2000+ with 768MB of RAM running Windows 2000 professional edition and is used only to simulate client requests. Server machine is a Pentium M 1.6MHz with 1024MB of RAM running JBoss application server version 3.2.6. The java virtual machine used to run JBoss is Java HotSpot Client Virtual Machine (client VM) version 1.5.0. JVM’s 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 performed a warm-up composed of one client executing 10,000 requests in a row.

To validate the designed models, JBoss source code is instrumented in order to register the number of bean instances necessary to process clients’ requests and the instant in time in which each one is created.

Once the environment is completely isolated and prepared, JBoss is started and the benchmark application is deployed. This application is set up to use the default instance pooling configuration (minimal size = 0, maximum size = 100, operation mode = non-strict). Finally, each one of the described workloads is repeatedly submitted.

After that, TimeNET [Zimmermann 2001] version 3.0.4 (for Linux platform) is used to verify the behavior of the same metrics (number of created instances and throughput) in the GSPN models. Workloads similar to those used in measurement experiments are applied to both models (base and refined) during the same 100 seconds. During these experiments, TimeNET’s transient simulation function was set up with the following parameters: Number of sampling points = 100, Confidence level = 95% and Maximum relative error = 10%.

![Figure 8. Instance Creation: 5 Clients at 100 Req/s.](image)

After that, TimeNET [Zimmermann 2001] version 3.0.4 (for Linux platform) is used to verify the behavior of the same metrics (number of created instances and throughput) in the GSPN models. Workloads similar to those used in measurement experiments are applied to both models (base and refined) during the same 100 seconds. During these experiments, TimeNET’s transient simulation function was set up with the following parameters: Number of sampling points = 100, Confidence level = 95% and Maximum relative error = 10%.
Figure 8, 9, 10 and 11 show the number of instances created in simulations performed using base and refined models as well as in measurement experiments. In spite of the variations observed in some experiments during the initial moments, the simulation results obtained from the designed GSPN models are reasonable accurate. In general, the GSPN refined model better approximates the measured values when compared to the GSPN base model. However, refined model has more states and transitions. As a consequence, its computational costs are greater and corresponding simulation experiments take more time to conclude. As the results obtained from both models are not far from one another, experiments performed with base model give a very good approximation and are good enough for most purposes, like capacity planning.
The results also demonstrate that, in all the considered scenarios, the number of created instances levels off. This is a very important information since it indicates that, in those scenarios, the resource consumption is limited. Another particularly relevant result can be seen in Figure 11. That figure represents a scenario involving 100 clients, each one carrying out 1 request per second. Despite the number of clients considered, a very small number of instances are created to process their requests. This scenario clearly demonstrates the effect of the pooling mechanism.

![Figure 11. Instance Creation: 100 Clients at 1 Req./s](image)

Finally, it is worth observing that, in all the analyzed scenarios, the fraction of the instance pool which is effectively used is very small. In fact, although the instance pool capacity has been configured to store a maximum 100 instances, none of the scenarios required the creation of more than eight.

![Figure 12. Overall Throughput for 5, 10 and 15 Simultaneous Users at 100Req/s.](image)
Figure 12 presents throughput results for three scenarios mentioned in the beginning of Session 5. The first one comprises 5 users, each one performing 100 req./s what correspond to a nominal overall throughput of 500 req/s. The second one comprises 10 users, also at 100 req/s, corresponding to a nominal throughput of 1,000 req/s. The last one comprises 15 users for an nominal throughput of 1,500 req/s. Results presented in Figure 12 make clear that JBoss server is not representing a bottleneck to entire system, as far as it is processing requests as fast as they are incoming, i.e. JBoss server is operating nearly at the scenario’s nominal throughput.

6. Conclusions

This paper has investigated the use of stochastic Petri nets for performance prediction of application servers. To do so, performance models of the JBoss application server are designed and simulation and measurement experiments are carried out.

The simulation results obtained from the designed GSPN models are very close to the results obtained by the use of the measurement technique, validating both, the approach and the models themselves. Once the execution of measurement experiments is not an easy task, having an accurate model to perform simulation experiments is clearly a very important advantage.

Despite of the relevance of the results obtained, the proposed models have some limitations. Firstly, those models can not be used for performance prediction in scenarios involving other kind of components (for example, entity beans). Secondly, time spent in simulation experiments depends heavily on the number of clients considered. As a consequence, time can be a limiting factor to simulations of scenarios involving a large number of clients. Finally, the low level of abstraction of the proposed models restricts the use of analytical techniques to scenarios involving a small number of clients. This problem is known as space state explosion.

Future works will consider the development of higher level models, having as a result the immediate and easy use of analytical techniques to carry out performance analysis.

7. References


