Performance Models for the Instance Pooling Mechanism of the JBoss Application Server

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

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

Abstract

The capacity management of a Java Enterprise Edition (JEE) application involves the tuning of numerous parameters at different layers that include the database management system, network, operating system, Java Virtual Machine, JEE application server and application code level itself. At the application server level, an important element to be considered is the instance pooling mechanism, which manages the creation of instances required to process client requests. This paper proposes Petri net models to represent the instance pooling mechanism embedded in the JBoss application server and demonstrates how those models can be used for performance evaluation.

1. Introduction

Java Enterprise Edition (JEE) platform is one of the most prominent technologies used to develop enterprise applications. The mentioned success is largely related to its openness and to the rich set of APIs that it offers. Different vendors offer implementations of JEE platform which are referred to as JEE application servers.

The core of JEE specification is the definition of a framework for the development of server-side components known as Enterprise JavaBeans (EJB). EJB components are deployed in an application server and hosted in a runtime environment, named EJB container, which is responsible for managing their life cycle and for providing them with system-level services such as transaction management, concurrency and security. To provide these services in a transparent way, an application server acts as a proxy intercepting requests and interposing the container between the clients submitting requests and the components responsible for processing them.

Another service cited by the EJB specification is instance pooling. To optimize performance, an EJB container usually associates an instance pool with each component that it is responsible for. Once the container intercepts a request directed to a component, it asks the instance pool for an instance. Then, the pool may create a new instance to process the request, or otherwise, if there is an idle instance in memory, it may reuse it, reducing the time and memory necessary to process the request. Sometimes, an instance pool may also reduce allocated resources by destroying instances that have not been used recently. The mechanism used to perform instance pooling is not part of the EJB specification, but depends on the vendor implementation.

Despite the facilities offered by JEE platform, developing and maintaining large and scalable business applications represent a big challenge. Under heavy demand those applications can consume a large amount of resources from the server environment and the lack of those resources can degrade their performance. In fact, a business application can have a large number of concurrent users, thus, a huge number of objects can be simultaneously in use. In this scenario, in addition to the great amount of memory used, concurrency and transactional concerns can meaningfully degrade the throughput and response time of that application.

A strategy to cope with a heavy demanding scenario is to throttle the amount of work that an EJB server can perform concurrently. One way to do that is by limiting the size of the thread pool, and consequently specifying an upper bound to the number of requests that can be simultaneously processed. That approach does not consider the class of the requests and affects the server performance as a whole. Another solution is to limit the size of the instance pools. This tuning can be done on an EJB component level basis, allowing, for example, more requests to execute with lightweight beans and fewer with heavyweight ones. That flexibility demonstrates the importance of the instance pooling mechanism to the overall performance of an EJB server.
To study the impact of the pooling mechanism, this paper presents a case study focused on the JBoss Application Server and proposes Generalized Stochastic Petri Net (GSPN) models which are used in performance experiments.

2. Performance evaluation

Performance evaluation of software system is usually carried out through the adoption of two techniques: measurement and performance modelling.

The measurement technique has been widely used in the performance evaluation of application servers [1][2][8][15]. It encompasses the planning and execution of measurement experiments directly in the system under study. In these experiments, workload generation tools are used to emulate real users defining a synthetic workload that can be applied repeatedly. Meanwhile, monitoring tools are used to gather, analyze and present data related to a set of metrics considered relevant. The major drawback of the measurement technique is that it can only be used when the system already exists. Besides, the costs associated with configuring an environment and acquiring the necessary tools can be prohibitive.

On the other hand, the performance modelling technique does not require that the real system is already working. Instead, it proposes the construction of models, which are abstract representations of the system. Typically, those models embed information obtained directly from the system itself, from its documentation and/or from published results. Two types of models can be developed: simulation models and analytical models.

A simulation model consists of a computer program that mimics the operation of the system. In those models, the system state is usually described by a set of discrete variables and its operation is represented by a chronological sequence of events, each one denoting a change of state. During a simulation, a number of counters accumulate statistics usually related to the time spent in the system states. Those statistical data are used to estimate the values of important performance metrics. Generally, a simulation should be run until each metric can be accurately computed (as stated by the confidence level and the width of the confidence interval desired). Therefore, it is important to realize that the size of the simulation model, the accuracy of the results and the number of metrics considered have a great impact on the duration of the experiment. Furthermore, due to the level of detail that is needed to build accurate simulation models, they can be expensive to develop and hard to validate. On the other hand, once built and validated, simulation models allow for the investigation of the system at a detailed level. Recently, expressive results in the performance evaluation of distributed systems have been obtained by using simulation models [6][7][12].

An analytical model represents a system through a set of mathematical expressions that can be solved to derive the desired performance metrics. Frequently, in order for an analytical model to be tractable, it must embed a number of simplifications and represent the system in a high level of abstraction. Hence, those models should be hardly detailed and the accuracy of the results obtained can be compromised. On the other hand, analytical models are regularly quite fast. Some important analytical models have been proposed to enable the performance evaluation of application servers [4][5][9][10].

The tradeoffs involved in the selection of a modelling approach suggest that an interesting aspect consider while selecting a modelling tool is its ability to support the development of models that can potentially be solved by both, simulation and mathematical analysis. In this context, Stochastic Petri Net (SPN) models are particularly attractive since, beside enabling the use of both techniques, they offer a graphical representation of the modelled system and have support to denote important aspects such as concurrency, conflict, synchronization and mutual exclusion.

SPN models are based in the idea that a system in execution assumes different states that are modified by the occurrence of events. Those models represent the states by places and tokens, and the events by transitions associated with exponential delays. Generalized Stochastic Petri Net [11] proposes an extension to the SPN and includes the concept of immediate transitions, which are used to represent events that consume no time (or a very little time) to execute. Considering those aspects, this paper elected GSPN as the modelling tool of choice.

3. JBoss application server

JBoss Application Server [3], an open-source JEE implementation supported by Red Hat, is one of the most influential JEE products available on the market. This success and the availability of its source code were decisive factors that lead to the selection of this server for a case study. The subsections bellow present the aspects of the JBoss implementation that are considered relevant to the understanding of the models proposed in this paper.

3.1. EJB and JBoss

According to the JEE specification, an EJB component must be deployed and run inside a EJB container that is responsible for offering it a set of services. To plug these services in a transparent way, an application server ought to implement a proxy which intercepts the requests and interposes the container between the client application and the component.

On the architecture of the JBoss, the proxy has a chain of client-side interceptors, each one being responsible for
collecting the data necessary to the implementation of a service (e.g. a security client-side interceptor would collect data related to the current user) and for forwarding the request to the next one in the chain. The last interceptor in that sequence is in charge of dispatching the request and the data collected (referred together as an invocation) to the invoker proxy, which marshals and forwards that invocation through the network.

At the server side, an element named invoker is responsible for receiving, unmarshalling and forwarding the invocation to the container that hosts the required EJB component. Next, the container forwards the invocation to a chain of server-side interceptors, each one responsible for implementing a specific service. Each interceptor reads the data collected by the corresponding client-side interceptor and uses that data to provide its service. After that, an interceptor forwards the invocation to the next one in the chain. Eventually, the instance interceptor, which is responsible for implementing the instance pooling mechanism, is invoked and it obtains the instance of the component that will be used to process the request. When the last interceptor is invoked, it identifies the method that was originally requested by the user, and invokes it in the instance obtained by the instance interceptor. If the method returns a value, it transverses the chain in the reverse way until it is received by the container which delivers the returned value to the invoker. Afterwards, the invoker marshals that value and sends it back to the invoker proxy, which unmarshals and forwards it through the client-side chain in the reverse way. Finally, the returned value is forwarded to the client application. Figure 1 shows the key elements of that architecture.

### 3.2. The instance pooling mechanism

According to the EJB specification, an important service provided by a container to a component is the instance pooling. However, the precise algorithm used to implement that service is not described by the EJB specification itself, so it is considered vendor specific.

Particularly, JBoss implements a very flexible instance pooling mechanism that can be completely configured, for each individual component, by using an XML file. The configurable parameters include maximum size, minimum size and operation mode. Maximum/minimum size is the maximum/minimum number of instances of that component that the pool can store (default values are 100 and 0, respectively). By the time the first request for that component comes in, the instance pool creates and initializes the minimum number of instances configured.

The instance pooling mechanism of the JBoss server can operate in two different modes: strict and non-strict. In non-strict mode, the instance pool can create any number of instances of the component in order to process simultaneous requests. Thus, if the number of instances created is greater than the maximum size configured to the pool, the additional instances will be garbage collected. In the strict mode, the number of instances created is limited by the maximum size of the instance pool (no matter how many simultaneous requests the component receives).

In summary, the instance pooling mechanism operates as follows: when the instance interceptor (see Section 3.1) receives an incoming request, it must require an available instance from the corresponding instance pool. At this point, if the pool has an available instance, it is returned, 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 behavior 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 instance back to the instance pool, which clears any information associated with the previous request. If the instance pool is not full, it can retain that instance. Otherwise, the instance becomes eligible to be garbage collected.

### 4. GSPN models

This section presents two GSPN models for the JBoss Application Server. These models are an evolution of the models presented in [14]. The intent of the models is to enable the estimation of some performance metrics concerning the server operation under a projected workload. The set of metrics includes CPU utilization, throughput and instance creation, which is specially related to the instance pooling mechanism and gives an idea about memory consumption.

The proposed models represent a scenario composed of a set of clients carrying out HTTP requests following a
Poisson distribution 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 stateless session bean and some web components. Clients access this application by sending HTTP requests that are processed by the web components. These web components invoke an instance of an EJB component that returns a string. The information returned is placed in an HTML page that is sent back to the client.

4.1. Client/Server model

The first model, depicted in Figure 2 and referred to as Client/Server model, describes in detail what happens when the JBoss server receives a request to an EJB component. Particularly, it focuses on the instance pooling mechanism and enables a comprehensive investigation of the core aspects related to it. A set of timed transitions represent the major steps performed by the server during a request processing. Mean values calculated from the results of some measurement experiments performed in an instrumented version of the JBoss approximate the delays associated to these transitions. The proposed model represents three main elements: clients, network and JBoss application server.

In the Client/Server model, clients are represented by the subnet comprising the places ClientsReady and SendingRequest, the 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, the transition generateRequest is exponential and has infinite server semantics. Once a request is generated, a token is put in place SendingRequest indicating that this request is now ready to be sent over the network. The delay of generateRequest transition (dRequest) is a variable parameter that may be changed in order to represent different request rates.

The network is represented by a subnet comprising two timed transitions, namely sendRequest and sendResponse, with exponential delays and single server semantics. The delay parameters can be approximated based on the size of the data that is exchanged between the client and the server.

The JBoss server is modeled by a composition of four components: PreviousInterceptors, InstanceInterceptor, InstancePool and NextInterceptor. These components
share access to the CPUs of the server machine, 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 by those immediate transitions whose names start with sync. After acquiring a CPU, a component enters in a local state (represented by a place whose name starts with Waiting) and remains in this state until the corresponding task is finished and the CPU is released. Since the focus of this paper is to represent and investigate the instance pooling mechanism, the other services provided by the server are represented in a more abstract way.

The PreviousInterceptors component is responsible for receiving and forwarding the incoming requests and for sending the outgoing responses. The received requests are represented by tokens in RequestsReceived. A token in that place enables the transition invokeInChain that represents all tasks performed by the server-side interceptors preceding the invocation of the InstanceInterceptor. When that transition is fired, the request is forwarded to the InstanceInterceptor which asks the InstancePool for an available instance and forwards the request for further processing. Eventually, the EJB instance will process the request and send a response. That response crosses the chain of interceptors in the reverse way. Again, the work carried out by the interceptors is represented by a single transition called outChain. When this transition fires, a token is created in SendingResponse indicating that the outgoing response is now ready to be sent back to the client.

The InstancePool acts as a repository of instances. After receiving a request, the InstancePool logs it and searches for an available instance in place Pool. This search is indicated by the place CheckingPool and the transitions poolIsEmpty and poolIsNotEmpty. If there are idle instances in the pool (i.e. #Pool > 0), poolIsNotEmpty gets enabled and fires, generating a token in WaitingGet. This token enables the getPooled transition that represents the obtaining of a pooled instance. Otherwise (i.e. #Pool = 0), the transition poolIsEmpty fires and a token is moved to WaitingCreate. This means that a new instance must be created. In this model, instance creation is represented by the transition create. Each created instance is counted by putting a token in NrCreated. To structurally limit the Petri net, the place CreateLimit and the transition T1 are represented. T1 has a great delay, so it does not affect the instance creation counter. Eventually, a token is put in the place InstanceReceived representing that an instance is available to the InstanceInterceptor.

After obtaining an instance, the InstanceInterceptor calls the NextInterceptor which has a single transition, invokeNext, to represent the tasks performed by the remaining interceptors and the method of the EJB component. When invokeNext fires, the instance used is returned to the InstanceInterceptor and a token is put in NextInterceptorFinished. Finally, the InstanceInterceptor can return that instance to the InstancePool.

When the InstancePool receives an instance, it erases all the information associated with the previous request. This task is modeled by the transition clear. After that, the pool checks its current and maximum size in order to decide if that instance should be discarded or be retained for further reuse. This decision is modeled by the places ReleasingBean and FreeSlots, and the transitions discard and goToPut.

### Table 1. Mean (µ), standard deviation (σ) - Client/Server model.

<table>
<thead>
<tr>
<th>Transition</th>
<th>µ (µs)</th>
<th>σ (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>2.147</td>
<td>0.568</td>
</tr>
<tr>
<td>create</td>
<td>187.115</td>
<td>477.775</td>
</tr>
<tr>
<td>getPooled</td>
<td>1.967</td>
<td>0.521</td>
</tr>
<tr>
<td>invokeInChain</td>
<td>212.198</td>
<td>805.235</td>
</tr>
<tr>
<td>invokeNext</td>
<td>5.607</td>
<td>88.007</td>
</tr>
<tr>
<td>log</td>
<td>2.274</td>
<td>2.413</td>
</tr>
<tr>
<td>outChain</td>
<td>24.739</td>
<td>98.401</td>
</tr>
<tr>
<td>put</td>
<td>1.985</td>
<td>0.552</td>
</tr>
</tbody>
</table>

To complete the description of the model, it is necessary to adjust the delay of the modeled transitions. This step involved carrying out six experiments conducted to measure the delays associated with the activities represented by the transitions. Each experiment consisted of a unique client executing 50,000 requests to a benchmark application (see Section 5). The first 10,000 were intended to warm-up the JBoss server and the corresponding JVM and were discarded. In order to minimize interference between two consecutive experiments, the JBoss server was restarted at the end of each one. After those experiments, the mean and the standard-deviation were determined for each measured delay. The data collected are summarized in Table 1.

### Table 2. Timed transitions - Client/Server model.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Delay in µs</th>
<th>Firing Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>2.147</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>create</td>
<td>187.115</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>getPooled</td>
<td>1.967</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>invokeInChain</td>
<td>212.198</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>invokeNext</td>
<td>5.607</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>log</td>
<td>2.274</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>outChain</td>
<td>24.739</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>put</td>
<td>1.985</td>
<td>Infinite-Server</td>
</tr>
<tr>
<td>sendRequest</td>
<td>24.508</td>
<td>Single-Server</td>
</tr>
<tr>
<td>sendResponse</td>
<td>59.668</td>
<td>Single-Server</td>
</tr>
</tbody>
</table>

The details concerning the timed transitions are summarized in Table 2. The delays of the sendRequest and
sendResponse are calculated by using the size of the request and response packets, the network bandwidth and the mean time needed to open and close connections which was measured by using the Ethereal tool.

4.2. Abstract model

While the Client/Server Model is very useful and may be simulated, it is not amenable to be solved analytically (specially for a large number of clients). Particularly, the viability of solving a Petri net model analytically depends on the number of states achievable by the system, which varies exponentially with the number of places.

That limitation has lead to the proposition of the Abstract model (see Figure 3), which represents the same scenario in a more abstract level. As the previous model, this one assumes that the instance pooling mechanism is operating in non-strict mode. An additional assumption is that the instance pool is initially full, so that it is only necessary to create a new instance when the number of clients is greater than the pool size. In this case, the instance becomes eligible to be garbage collected once the request processing is finished. In scenarios with a small number of clients this assumption basically means that the system has been operating for some time so that the transient behaviour cannot be correctly analyzed. Particularly, it means that is not possible to investigate instance creation over time anymore.

![Figure 3. Abstract model.](image)

The Abstract model has two components: the Client and the Server.

The Client component comprises one place, ClientsReady, which stores tokens representing the clients ready to perform requests, and one transition, generateRequest, whose delay, dRequest, represents the interval between successive requests submitted by a client. This transition is distributed exponentially, meaning that the generation of requests follows a Poison distribution, and has an infinite-server firing semantic, enabling concurrent clients to send requests simultaneously.

The Server component represents both the network and the JBoss server. It has two exponential transitions, systemWGet and systemWCreate, two immediate transitions, doGet and doCreate, and four places, WaitingCPU, ProcessingCreate, ProcessingGet and CPUs.

The systemWGet transition represents the behaviour of the system when it receives a request and there is an idle instance in the pool which can be used to process it. The transition systemWCreate describes the behaviour of the system when a request is received and the instance pool is empty, i.e., it is necessary to create a new instance to process it. The delays of the systemWGet (dSystemWGet) and systemWCreate (dSystemWCreate) transitions were determined by calculating the mean of the values obtained in the same measurement experiments used to calibrate the Client/Server model and were 335,093 and 518,257 microseconds, respectively.

The WaitingCPU place receives tokens modelling the requests sent by the clients. To process these requests, it is necessary to allocate a CPU which is represented by a token in CPUs. When a CPU becomes available, the immediate transitions doGet and doCreate become enabled and the conflict between them is solved by using weights associated with each one. To assign an adequate value to each weight, it is necessary to analyze when those transitions should be fired.

Since the pool is initially full of idle instances, the systemWCreate transition should be fired only when the number of clients is greater than the pool size and the request generation rate is greater than system processing rate. Otherwise, the incoming requests are processed by using instances available in the pool, meaning that the systemWGet transition ought to be fired. Table 3 summarizes the weights used in the model, where nClients represents the number of clients, nClients/dRequest is the request generation rate and 1/dSystemWGet is the system processing rate.

### Table 3. Immediate transitions – Abstract model.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>doGet</td>
<td>IF(nClients/dRequest&gt;=(1/dSystemWGet)) AND (nClients&gt;poolSize) poolSize</td>
</tr>
<tr>
<td></td>
<td>ELSE 1000000000000000;</td>
</tr>
<tr>
<td>doCreate</td>
<td>IF(nClients/dRequest&gt;=(1/dSystemWGet)) poolSize</td>
</tr>
<tr>
<td></td>
<td>ELSE nClients - poolSize 0.00000000000001;</td>
</tr>
</tbody>
</table>

Table 3 indicates that if the request generation rate is greater than the request processing one and the number of clients is greater than the pool size, the firing rate of the doGet transition is proportional to the poolSize, whereas the firing rate of the doCreate is proportional to nClients - poolSize. In other situations, the weight associated to the doGet transition is especially big while the weight of the
doCreate transition is insignificant, indicating that the incoming requests are processed by using the instances available in the pool.

5. Experiments and results

The benchmark application used to calibrate and validate the models comprises a stateless session bean and some web components. The clients access this application sending HTTP requests 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. To validate the proposed models, the benchmark application was set to use the default instance pool configuration which has a minimal size of 0 and maximum size of 100 and operates in non-strict mode.

5.1. Client/Server model

The validation of the Client/Server model involves the performance of measurements and simulations, and the comparison of the obtained results. According to [], differences in resource utilization and throughput metrics of about 10% are considered acceptable. In this context, three scenarios are considered. The first two represent 5 and 10 clients carrying out 100 requests per second. The third scenario represents a more realistic workload and is composed of 100 clients, each one performing 1 request per second. Also, three metrics are considered: number of created instances, throughput and CPU utilization.

The simulations were performed by using the TimeNet tool with a confidence level of 95% and a maximum relative error of 10%. The measurements were performed in an isolated Fast-Ethernet network containing only two machines: a client and a server. The client machine was an Athlon 2000+ with 768MB of RAM running Windows 2000 Professional. The server machine was a Pentium M 1.6MHz with 750MB of RAM running a JBoss application server version 3.2.6. During the experiments, no useless applications and services were running. A warm-up sequence, comprising one client executing 10,000 requests, was performed before each experiment. For each scenario, 35 measurement experiments were carried out.

5.1.1 Scenario 1: 5 clients – 100 requests/s. In this scenario, the clients perform requests very quickly and the server needs practically one instance to process each client. Figure 4 shows the number of instances created in measurement and simulation experiments.

As can be observed, the measurement and simulation results are close. In fact, by the end of 100 seconds, the measurement experiments led to a mean of 4.97 instances while the simulations led to 4.32 instances, corresponding to an error of 13.07%.

In addition, the measurements indicated that the mean value of throughput was 499.77 req/s, while the simulations indicated 481.28 req/s, which corresponds to an error of 3.70%. These results show that the server was not saturated, since it was able to process the requests almost in the same rate that they were generated.

5.1.2 Scenario 2: 10 clients – 100 requests/s. In this scenario, the number of clients duplicated in relation to the previous one. Figure 6 shows the number of instances created in measurement and simulation experiments. In spite of the differences verified in the beginning of the observation period, the lines representing the measurements and simulations seem to converge. In fact, the results of the measurement experiments pointed out that 6.89 instances were needed to process the requests.

The third metric considered was CPU utilization. The Performance Monitor tool was used to collect the needed data. The results of the measurement and simulation experiments are presented in Figure 5.

The measurement experiments pointed out that the mean value of the CPU utilization was 17.89%, while the simulations pointed out a mean value of 11.85%. Despite the difference between these values, they indicate that the CPU was not a bottleneck in this scenario.

Since the number of clients was quite small, it was possible to solve this model analytically. The calculated solution indicated a total of 4.73 created instances, a throughput of 481.59 req/s and a CPU utilization of 12.08%.
while the simulations pointed out a total of 6.68 instances, corresponding to an error of 3.04%.

In terms of throughput, the results of the measurement experiments pointed out a value of 999.85 req/s, while simulations indicated 956.47 req/s, corresponding to an error of 4.34%.

![Figure 6. Scenario 2: number of instances.](image)

![Figure 7. Scenario 2: CPU utilization (%).](image)

Finally, Figure 7 represents the CPU utilization metric. Despite some initial differences, the results seem to indicate a trend of convergence between measurements and simulations. In terms of mean values, measurements pointed out an utilization of 31.97%, while simulations indicated 23.67%. Given the number of clients, it was not possible to obtain an analytical solution to this scenario.

5.1.3 Scenario 3: 100 clients – 1 request/s. This scenario models a more realistic situation with a greater number of users and a smaller request rate. As Figure 8 shows, measurements and simulations indicated that despite the large number of clients, just a small number of instances were created. In fact, simulations pointed out a mean of 3.14 instances while measurements pointed out 2.54. These results show that a pool with a maximum size of 10 instances is enough to support that workload.

Regarding the throughput, measurements indicated a mean value of 99.99 req/s, while simulations indicated 99.97 req/s, i.e., the difference is 0.02%.

Finally, Figure 9 shows the data gathered for the CPU utilization in the measurement and simulation experiments. The curves show that simulation and measurement results are relatively close. A detailed analysis indicates that those curves are almost all the time below 5%, pointing out a small utilization of the CPU.

![Figure 8. Scenario 3: number of instances.](image)

![Figure 9. Scenario 3: CPU utilization (%).](image)

5.2. Abstract model

The scenarios and metrics used to validate the Client/Server model have also been used to validate the Abstract one. However, the results of the abstract model were obtained analytically through the solution of the corresponding Markov chain. To reduce the number of places and enable the obtaining of an analytical solution, this model does not represent the pool and the instances explicitly, but through the use of the poolSize parameter. Consequently, it is assumed that the pool is already full of instances and only steady state solutions can be derived. Thus, the metric concerning the number of created instances is not considered.

Table 4 summarizes the results obtained through the measurements, the analytical solution of the Abstract model and the simulations of the Client/Server model.

Although the throughput and number of created instances obtained through the Client/Server model are very close to those gathered from the measurement experiments, the results obtained to the CPU utilization metric are not quite close. In fact, while the Client/Server model apparently capture the contention for the CPU, it does not accurately represent all the software contention.
aspects. An attempt to do that would lead to a very large model which could be useless to performance evaluation.

On the other hand, all the results calculated from the Abstract model are very close to the measurement results demonstrating its accuracy and encouraging the construction of high-level models.

Table 4. Scenarios and results.

<table>
<thead>
<tr>
<th>Clients</th>
<th>Requests (req/s)</th>
<th>Experiment</th>
<th>Throughput (req/s)</th>
<th>CPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>100</td>
<td>Measurement</td>
<td>499.77</td>
<td>17.89</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>Client/Server</td>
<td>481.59</td>
<td>12.08</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>Abstract</td>
<td>481.56</td>
<td>16.14</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>Measurement</td>
<td>998.85</td>
<td>31.97</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>Client/Server</td>
<td>956.35</td>
<td>23.98</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>Abstract</td>
<td>955.69</td>
<td>32.02</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>Measurement</td>
<td>99.97</td>
<td>3.87</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>Client/Server</td>
<td>99.97</td>
<td>2.50</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>Abstract</td>
<td>99.97</td>
<td>3.35</td>
</tr>
</tbody>
</table>

6. Conclusions

This paper discusses the importance of the instance pooling service to the performance of an application server and presents a case study based on JBoss. Particularly, it proposes Petri net models that represent this server and highlight the embedded pooling mechanism.

Two performance models are proposed: a detailed one (referred to as Client/Server), which is intended to be used in simulations, and an abstract one, which is intended to be solved analytically.

To validate those models, three scenarios are proposed. The first two represent a few high demanding clients, while the third one represents a more realistic situation comprising a greater number of clients submitting requests at a smaller rate. Furthermore, three metrics are considered: number of instances, throughput and CPU utilization.

In general, the results obtained by using simulation and analysis techniques are close to those obtained from the measurement experiments, giving an indication of the accuracy of the proposed models.

Once the execution of measurement experiments is not an easy task, having accurate models to predict performance 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 kinds of components (e.g. databases and queues). 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, software contention is not fully represented and that can affect the prediction of the saturation point.

References